React
前端技术与工程实践

李晋华　编著
王桂强　杨甫勤　审校

電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

内容简介

本书是一本专门介绍 React 前端框架基本原理及其相关工程实践的技术参考书。全书分为 14 章，主要包括 React 技术基本原理、相关前端开发工具链、实用技巧及热门资源介绍四部分。全书结构完整、层次清晰，由浅入深地介绍了 React 前端技术的原理、相关工具链的使用及 React 技术在工程中的应用技巧等。本书关注技术原理，在讲解技术应用的同时介绍相关原理和理念，帮助读者更深入地理解和掌握 React 技术，并能尽快地投入实际应用。本书也尽可能全面地囊括当前 JavaScript 前端工程开发的相关技术与工具，通过本书可以全面地掌握 React 相关的知识体系并较快地进入实际工程开发。本书语言浅显易懂，辅以生动的实例，是 React 前端工程开发的好助手和好工具。

本书适用于对前端开发有一定了解和开发经验的读者，也可作为相关培训教材使用。

图书在版编目（CIP）数据

React 前端技术与工程实践 / 李晋华编著. —北京：电子工业出版社，2017.4
ISBN 978-7-121-31050-8

Ⅰ. ①R… Ⅱ. ①李… Ⅲ. ①移动终端 – 应用程序 – 程序设计 Ⅳ. ①TN929.53

中国版本图书馆 CIP 数据核字（2017）第 044537 号

策划编辑：孙学瑛
责任编辑：徐津平
特约编辑：赵树刚
印　　刷：三河市双峰印刷装订有限公司
装　　订：三河市双峰印刷装订有限公司
出版发行：电子工业出版社
　　北京市海淀区万寿路 173 信箱　　邮编：100036
开　　本：787×980　1/16　　印张：17.5　　字数：308 千字
版　　次：2017 年 4 月第 1 版
印　　次：2017 年 4 月第 1 次印刷
定　　价：69.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式：010-51260888-819，faq@phei.com.cn。

前　言

随着 AJAX 技术、Chrome JavaScript V8 引擎的出现，以及移动端的兴起，前端发生了天翻地覆的变化。传统的 JavaScript 知识体系即将过时。前端正以全新的思路和革新的理念得到越来越多的重视和关注，涌现出了众多技术，如 Node.js、NPM、CommonJS、AMD、ES 6、Webpack、Babel、React、AngularJS 等。众多新技术的冲击让人无所适从，而且，往往一个技术会附带一系列相关的技术和工具，更是让人难以下手。

React 技术的更新发展也很快，相关的资料在网上虽然能查到，但往往只是针对一个特定技术点，语焉不详；或者使用了最新语法，读者难以理解；又或者文档与软件版本不匹配，导致在模块安装时出现莫名提示、运行错误等问题。

有感于此，编者编著了本书，针对这些痛点，力图以 React 技术为抓手，整体介绍与当前前端相关的主流技术体系与使用途径，为读者找到一条技术的主干脉络，方便读者全面快速地深入学习以 React 为代表的前沿前端技术。为使读者降低学习成本，并很快地投入到工程实践中，本书还介绍了前端开发环境搭建和相关工具链的使用，力图为读者呈现前端开发的全貌。另外，本书在逐层深入介绍 React 技术的同时，还少量地讲解了底层技术原理，方便读者深入理解。

本书内容

本书分四部分，第一部分讲解 React 的基本原理和架构。考虑到相关辅助工具对知识的干扰，此部分排除外围技术干扰，以最朴素、最原始的方式来看 React 的

本质和原理，同时针对实际应用场景介绍了典型组件的开发思路及代码。第二部分讲解 React 相关工具链的原理和使用方法，切入面向工程化开发的前端开发技术体系，介绍相关工具的使用方法，并重点介绍与 React 相关的使用流程。第三部分讲解 React 的高级功能，如测试、路由等，是应对复杂界面的完整解决方案不可或缺的重要组成部分。第四部分介绍当前 React 的热门技术和相关资源。

源代码

本书的主要实例均附有源代码，源代码以实例包的形式发布在网上，读者可以自行下载。实例包中提供了 Node.js 的安装程序和运行说明文件。书中所提到的实例名对应网上同名文件夹。实例包根目录下的“使用说明.txt”文件说明了要运行的前提条件和实施步骤。

本书特点

- 新。本书中的 JavaScript 使用 ES 6 语法，React 针对 v15.0.0 以上版本，JSX 使用 Babel 6.x 版本等，确保读者掌握最前沿的知识和技能。
- 透。本书不是简单地介绍知识，而是透过知识来看本质的理念和原理，只有这样才能把技术吃透、用活。
- 全。本书力图将 React 相关的技术体系集中在一起，包括 React 相关工具使用、React 实用技巧、React 高级框架等，使读者全面掌握 React，减少时间成本，提高知识获取效率。
- 实。本书的讲解和实例尽量向实际使用场景靠拢，所涉及的复杂组件实例（树形组件、分页组件、表格组件等）均可直接用于实际开发环境，且配有详细的解说，读者可以快速上手。

适用范围

（1）适用于从事前端技术开发且有一定 JavaScript 基础的初学者。
（2）适用于从事网站前端设计与制作的开发者。
（3）可作为相关培训机构的专题培训教材。

（4）可作为相关开发者的工具书。

本书约定

（1）在面向对象的语言中，成员函数也称为方法，本书统一称为“成员函数”或简称为“函数”。

（2）代码都具有阴影背景，以示区别。

（3）代码中要重点强调、提醒的部分使用粗体格式。

关于我们

参与本书编写的人员还包括韩岗、刘兰峥、胡松奇、刘彦君。尽管我们已经做了仔细校对，但书中疏漏和不足之处在所难免，如果在书中发现任何的文字和代码错误，非常欢迎读者朋友反馈给我们。如果您有好的建议、意见，或遇到与本书内容相关的疑难问题，都可以联系我们，我们会及时为您解答。服务邮箱：ljhiiii@sina.com。

目 录

第一篇 原汁原味的React

第二篇 React开发相关工具链

第三篇 React进阶

第四篇 React相关资源

第一篇

原汁原味的React

我们常看到的 React 介绍已经被 JSX、AMD、ES 2015 等繁华的技术所遮掩，其实 React 的核心是简单和简洁的。本篇力图揭开蒙在 React 表面的面纱，用最朴素的方式、最原生的 API 展现 React 的基本脉络，便于读者从本质特征上理解 React 的基本思想和方法，而这也往往是我们在实际开发中减少出错、排查问题的利器。

1

React简介

1.1 前端技术发展及趋势

我们正处于一个 Web 前端技术变革的时代。早期嵌入在浏览器中的 JavaScript 只是小众语言，性能也不好。但从 AJAX 技术出现以后，基于 JavaScript 的前端技术得到了前所未有的重视。而后 jQuery、Prototype、Dojo、ExtJS 等前端组件框架陆续出现，到 Google 公司专门为 JavaScript 研发的 V 8 引擎更使得 JavaScript 插上了腾飞的翅膀，前端生长日趋繁荣。

近年来移动应用的蓬勃发展，移动端的生长势头变得更加强盛。移动应用的出现给前端带来了许多新的挑战，如多终端适配问题、多分辨率适配问题、远程调试问题等，针对这些问题出现了各种解决方案，推动了 Web 技术的发展，Web 前端出

现了百花齐放的态势，Web 开发出现了新的变革。语言层面，出现了 CoffeeScript、TypeScript 等语言，对原来的 JavaScript 进行了语法增强，JavaScript 语言本身也出现了新的标准，如 ES 2015（也称 ES 6）等。Twitter 公司推出的 Bootstrap 试图从样式层面入手解决终端适配问题，出现了 LESS、SASS 和 Stylus 等预处理语言；在 JavaScript 模块管理方面，出现了 AMD、CMD、KMD 等多个模块管理规范，也衍生了 SeaJS、RequireJS 等模块化管理工具。包管理工具，经历了 components、bower、spm 后，npm 开始占据主导地位。在规范和标准上也有不少产出，Web Components 的出现给前端开发开辟了新思路；在 JS 调试方面，各浏览器提供了种类繁多、功能丰富的调试工具和方案；在自动化测试方面，PhantomJS 在自动化测试上逐渐取代了 Selenium，而 WebDriver 规范的出现进一步推动了自动化测试的进程；在构建工具上先后出现了 grunt、browserify、gulp、webpack、jspm 等，目前 webpack 逐渐成为了主流。

在前端框架方面，目前也出现了众多框架，其中的佼佼者 React 和 AngularJS 均出自名门公司，具有非凡的影响力和号召力。尤其是 React，它定位于前端组件化、高性能和跨平台，它提出的基于虚拟 DOM 的理念一出现就获得了广泛的关注和认可，也被 AngularJS 等其他前端框架所引入。

本书着眼于 React 技术，同时也对与 React 相关的 JS 工程化技术进行探讨。其目的是让读者从繁花缀眼的技术丛林中，找到一条平坦的探索之路并可直接用于实战。读者能从本书中获得最大的价值。

1.2 React简介

React 起源于 Facebook 的内部项目，该公司积极尝试引入 HTML 5 技术用来架设 Instagram 网站，过程中发现，对于复杂前端 HTML 5 性能下降明显，达不到预期效果。在经过对市场上所有 JavaScript MVC 框架调研后，都找不到能满足自己需

求的产品，于是决定自己开发一套。2013 年 5 月开发完成后就发布到开源平台上，一发布就引起了广泛的关注和认可。

React 的设计思想极其独特，是对前端技术的一大革命性创新，其性能优秀、代码逻辑简单、适用面广且能用于移动 APP 开发，受到越来越多的人的关注和使用，普通认为它代表了未来 Web 开发的主流方向。同时，React 关键性的虚拟 DOM 思想也陆续被其他框架引入，如 AngularJS2，这也进一步证明 React 理念的优秀和先进性。比如 GitHub 最新的源码编辑器 Atom 就是用 React 构建的；雅虎邮箱也正在使用 React 重构，等等。

ReactJS 官网地址：https://facebook.github.io/react/。

GitHub 地址：https://github.com/facebook/react。

1.3 React特点

1.3.1 虚拟DOM

在 Web 开发中，UI 界面总是需要根据数据生成对应的 DOM 由浏览器呈现出来，并随数据的变化而调整相应的 DOM，这就需要反复对 DOM 进行操作。复杂或频繁的 DOM 操作通常会对性能造成很大的影响。为此 React 引入了虚拟 DOM（Virtual DOM）机制：用户构建虚拟 DOM，由 React 将虚拟 DOM 渲染到浏览器 DOM 中。每次数据变化 React 都会扫描整个虚拟 DOM 树，自动计算与上次虚拟 DOM 的差异变化，然后针对需要变化的部分进行实际的浏览器 DOM 更新。虚拟 DOM 是内存数据，本身操作性能极高，对实际 DOM 进行操作的仅仅是差异变化，从而性能得到了很大的提高。

在保证性能的同时，用户不再需要关注某些数据的变化如何局部更新到一个或多个具体 DOM 元素，而只需要关心数据状态，以及对应数据状态下界面是如何渲

染的，除此之外的其他工作都由 React 自动高效地完成。这样既明晰了开发思路，又提高了开发效率。

由于我们的主要操作对象是虚拟 DOM，与真实浏览器无关，甚至是否是浏览器环境都没关系，只要存在从虚拟 DOM 到真实 DOM 的转换器，就可以实现虚拟 DOM 的最终界面呈现，从而达到跨平台的目的。而从虚拟 DOM 到真实 DOM 的转换工具由 react-dom 实现，从虚拟 DOM 到移动 APP 的转换工具由 react-native 实现。

1.3.2 组件化

虚拟 DOM 不仅带来了简化的 UI 开发逻辑，同时也带来了组件化开发的思想。所谓组件，即封装起来的具有独立功能的、可复用的 UI 部件。

经典 MVC 开发模式从技术角度纵向对 UI 进行划分，将视图、数据、控制器分离，实现界面呈现、数据、控制的分层架构，达到松耦合和复用的效果。对于 React 而言，则是从功能角度横向划分，将 UI 分解成不同组件，各组件都独立封装，整个 UI 是由一个个小组件构成的大组件，每个组件只关心自身的逻辑，彼此独立。

React 推荐以组件的方式去重新思考 UI 构成，将 UI 上每一个功能相对独立的模块定义成组件，然后将小的组件通过组合或者嵌套的方式构成更大的组件，最终完成整体 UI 的构建。例如，Facebook 的 instagram.com 整站都采用了 React 来开发，整个页面就是一个大的组件，其中包含嵌套的大量其他组件，读者有兴趣可以查看它背后的代码。

React 认为一个组件应该具有如下特征。

（1）可组合性（Composeable）：一个组件能够很方便地与其他组件一起配合使用，或者嵌套在另一个组件的内部。如果一个组件内部创建包含了另一个组件，那么说父组件拥有（own）它创建的子组件。通过这个特性，一个复杂的 UI 组件可以拆分成多个简单的 UI 组件。

（2）可重用性（Reusable）：每个组件都是具有独立功能的单元，它可以不加更改地在多个 UI 场景重复使用。

（3）可维护性（Maintainable）：每个小的组件仅仅包含自身的完整逻辑，更容易被理解和维护。

1.3.3 单向数据流

与 AngularJS 所提倡的双向数据绑定不同，React 设计者认为数据双向绑定虽然便捷但在复杂场景下副作用也很明显，相比之下 React 更倾向于单向的数据流动——从父节点传递到子节点，基于组件的设计使得组件逻辑简单而且更容易把握，组件只需要从父节点获取 prop 渲染即可。如果顶层组件的某个 prop 改变了，React 会递归地向下遍历整棵组件树并重新渲染所有使用这个属性的组件。

基于单向数据流设计，React 提出了单向数据流的架构模式 Flux。与 MVC 分层架构有所区别，Flux 以 M 层 Store 为核心，围绕 Store 设计 Action 和 Dispatcher，提供了管理数据的更高级别的框架。

当然，使用 ReactLink，React 也可以扩展为双向绑定，但不建议使用。

1.4 React与React Native

伴随 React 技术常常出现一个 React Native 的名词。React Native 是基于 React 技术的用来开发移动 APP 的框架。鉴于 React 优秀的设计理念，React Native 本质上也还是围绕虚拟 DOM 展开，只是在最终渲染时有所区别，React Native 将虚拟 DOM 不是渲染到浏览器而是渲染到对应的原生 APP 界面，这也是 React Native 的主要工作机制。虽然 React Native 与 React 中的组件并不完全一致，但思路是相通的。这也意味着我们基于虚拟 DOM 设计的前端代码结构与移动 APP 的前端代码结构基本相似，这也体现了 React 的设计理念“Learn once, write anywhere”。本书主要专注于基于 React 的 Web 开发技术，对 React Native 就不涉及了。

1.5 对React的几个认识误区

相比其他前端框架，React 在理念、定位、适用领域等方面均有所不同，容易产生一些认识上的误区，这里先澄清几点：

（1）React 不是一个完整的 MVC 框架，最多可以认为是 MVC 中的 V（View），甚至 React 并不非常认同 MVC 开发模式。但 React 可以扩展为 MVC 架构，如后文会提到 Router 和 Flux。

（2）React 虽具有服务器端渲染的能力，但其实现只能基于 Node.js 架构，目前还不能在其他平台上使用。因此服务器端渲染只能算是一个辅助功能，并不是其核心出发点。事实上，React 官方站点也几乎没有提及其在这方面的应用情况，本书也对这方面的内容忽略不提。

（3）React 不是一个新的模板语言，其用到的 JSX 也只是一个用于便捷描述的语法糖而已，不使用 JSX 的 React 也能正常工作。为了更清楚地理解 React 的基本原理，本书一开始并没有引入 JSX，而是直接使用原生 React 进行示例和讲解。

（4）有人经常拿 React 和 Web Component 相提并论，但两者并不是一个概念，Web Component 是一个前端的组件标准，React 可以理解为是 Web Component 的其中一个实现但也未做到完全符合这个标准。

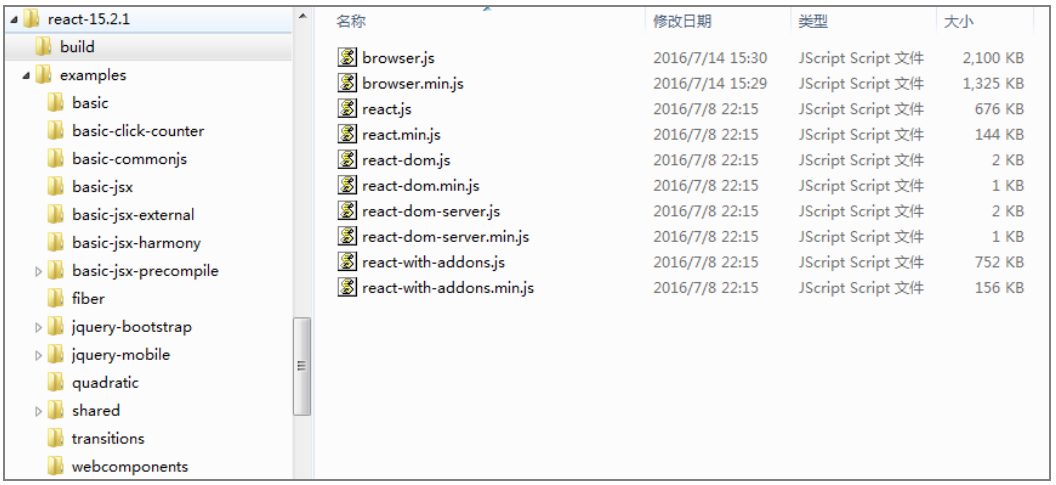
2

React基础

2.1 React最小环境搭建

如果不考虑工程化的问题，React 的运行基础环境非常简单，只需要在 HTML 文件中引入两个 js 文件（react.min.js 和 react-dom.min.js）即可开始工作。

以 Windows 操作系统为例（Linux 下类似），首先访问 https://facebook.github.io/react/，在页面中直接单击“Download React v15.2.1”按钮即可下载 React 最新版本（这里以 v15.2.1 版本为例），下载完成后的压缩包为 react-15.2.1.zip，解压后文件结构如下图所示。



其中，React 运行时最常用到的核心文件是 build 文件夹下的 react.min.js 和 react-dom.min.js 文件，均为经过代码压缩后的文件，通常部署时应使用这两个文件。这两个文件对应的未压缩版本文件是 react.js 和 react-dom.js，一般在开发时引用，便于调试。

建立工程文件夹，这里假定为 d:\react-projects。为方便描述，本书所有范例均以该文件夹作为相对路径的起点。

补充：

React 从 0.14 版本以上开始将 React 核心分成两个文件，这使得 React 结构更清晰。

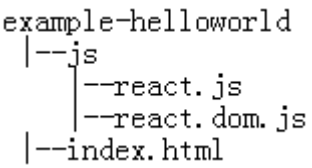
react.js 实现 React 核心的逻辑，且与具体的渲染引擎无关，从而可以跨平台共用。如果应用要迁移到 React Native，这一部分逻辑是不需要改变的。react.js 中包含了 React.createElement、React.createClass 等核心 API。

react-dom.js 则包含了具体的 DOM 渲染更新的逻辑，以及服务端渲染的逻辑，这部分就是与浏览器相关的部分了。如果要将应用迁移为移动 APP，则这部分可能会需要重新实现。react-dom.js 中主要的 API 是 ReactDOM.render。

React 支持的浏览器有 IE 8 以上版本、Chrome 和 Firefox 等。

2.2 Helloworld示例

为了直观感受 React 的魅力，按照“国际惯例”，我们先从最著名的 Helloworld 范例开始。在工程文件夹下建立第一个范例工程 example-helloworld。首先新建工程文件夹 example-helloworld，再按下图建立相应工程文件。



其中 index.html 内容如下：

```
<!DOCTYPE html>
<html>
  <body>
    <div id="reactContainer"> </div>

    <script src="js/react.js"></script>
    <script src="js/react-dom.js"></script>
    <script>
      var HelloComponent = React.createClass({
        render: function() {
          return React.createElement('h1',null,'Hello world');
        }
      });

      ReactDOM.render(
        React.createElement(HelloComponent,null),
```

```
        document.getElementById('reactContainer')
      );
    </script>
  </body>

</html>
```

至此已完成我们的首个 React 工程，这段代码仅仅实现了在浏览器窗口中输出字符串 Hello world，双击 index.html 可以直接在浏览器中看到效果。这个例子很简单，但千里之行，始于跬步，通过剖析这个示例，我们会了解到更多的信息。

可以看到，我们首先通过调用 React.createClass 方法注册了一个组件类 HelloComponent，这个组件类只包含了一个 render 函数，该函数通过调用 React.createElement 实现了以下 HTML 内容的输出：

```
<h1>Hello world</h1>
```

创建好的组件类 HelloComponent 随后被 ReactDOM.render 函数所调用。该函数将调用 React.createElement(HelloComponent,null)所生成的组件“挂接”到浏览器 DOM 中的 div 标签下，从而实现最终在浏览器的输出。

剖析这个过程，通过 React.createElement 调用产生的元素我们称为“虚拟 DOM”元素，与真实的浏览器 DOM 元素相区别。而这个元素及其所有子元素（如果有的话）最终要渲染到真实的浏览器 DOM 中才能得以呈现。虚拟 DOM 是 React 的核心思想，使用 React 只有两个步骤：一是创建虚拟 DOM（多层嵌套），二是渲染到浏览器中。在开发中我们的主要工作就是逐级创建出所需要的虚拟 DOM 树并响应用户的动作来改变虚拟 DOM 树的组成。

从这个例子中还可以感受到面向组件思想的体现，React 的操作围绕组件展开，使用组件需要先注册一个组件类，然后再实例化这个组件类。具体的功能实现均封装在组件中。

另外，还注意到，我们只是声明了组件的内容，并没有关注虚拟 DOM 怎么挂

接的问题，这些都由 React 替我们完成了，除此之外 React 还在幕后做了更多的工作，在后面的章节我们还会一一解读。

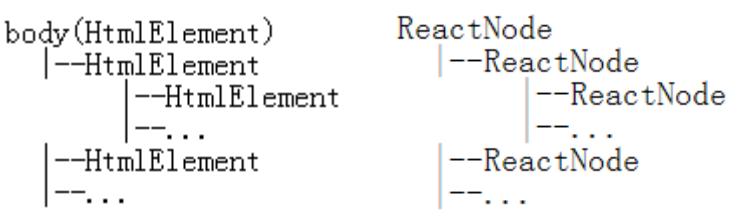
虚拟 DOM 是 React 的核心思想，理解这一点就把握住了 React 的本质。可以说，整个 React 技术均是围绕虚拟 DOM 而设计展开的，本书后面章节所涉及 React 的一些特性和特殊注意事项均根源于此。

2.3 React基本架构

2.3.1 虚拟DOM结构

React 为了更好的性能和跨浏览器兼容，实现了一个独立于浏览器的 DOM 和事件系统。浏览器 DOM 是由 HtmlElement 元素组成的树，虚拟 DOM 与浏览器 DOM 类似，也是由 ReactNode 节点元素组成的树，一个 ReactNode 实例表示一个轻量的、无状态的、不可变的虚拟 DOM 元素。ReactNode 与 HtmlElement 基本相似，只有微小的差别，而且 ReactNode 最终还是需要映射到 HtmlElement 才能展示出来。虚拟 DOM 树由 ReactDom.render 函数渲染到实际的浏览器 DOM 节点上，形成最终的界面效果。

两者之间的对比关系如下图所示。



ReactNode 的数据来源可以是字符串（string）、数字串（number）、ReactElement。

同时，ReactNode 也支持嵌套，可以包含子级 ReactNode 的数组。多级嵌套的 ReactNode 最终形成一颗虚拟 DOM 树。

用户对界面的操作事件首先由浏览器接收，如果是 React 渲染出来的 DOM，将会由 React 接管并派送到对应的 React 组件实例，并由组件实例对事件做出响应。

2.3.2 虚拟DOM元素

React 中最主要的类型就是 ReactElement，是 ReactNode 的主要数据来源。ReactElement 有四个属性：type、props、key 和 ref。除此之外，没有其他的属性或成员函数。

虚拟 DOM 元素的创建通常在 render 方法中完成。通过调用 React.createElement 创建 ReactElement 元素，createElement 方法的第一个参数就是 HTML 标签名或预先注册好的 ReactClass 对象；第二个参数是 ReactElement 的属性，使用 json 格式描述；第三个及以后的参数是该元素所要包含的子元素。参看下面的例子：

```
    var child1 = React.createElement('li', null, 'text1');
    var child2 = React.createElement('li', null, 'text2');
    var parent = React.createElement('ul', { className: 'myUlClass' },
child1,child2);
```

或：

```
    var parent = React.createElement('ul', { className: 'myUlClass' },
[child1,child2]);
```

注意：

对于自定义的组件类型名，React 约定首字母大写且需要预先注册。如果是 HTML 中规定的标签，则可以直接使用而不需要注册，但是标签名应该小写，React 通过组件类型名的首字母是否大写来判断是否自定义组件。

组件类型注册需要调用 React.createClass，调用 createClass 方法传递的参数是

组件类的属性和成员函数。

从上面的例子可以看出，更为复杂的虚拟 DOM 树也是通过不停地调用 React.createElement 方法构建出来的。

2.3.3 组件工厂

组件工厂是创建虚拟 DOM 元素的另外一种“简洁”语法，相当于一种快捷方式。下面的两行语句是等价的。

```
React.DOM.div()
```

等价于：

```
React.createElement('div')
```

前者使用的是工厂类方法，后者是普通创建方法。可以看出，使用工厂类方法创建组件只是相当于预置了第一个参数而已。

createFactory 函数用于创建工厂，其核心源代码如下：

```
function createFactory(type){
  return React.createElement.bind(null, type);
}
```

对于 HTML 标签来说，其工厂类已经预置。但对于自定义组件而言，首先需要为自定义的组件类创建一个工厂类，然后才能直接使用。

```
var divFactory = React.createFactory('div');
var root = divFactory({ className: 'my-div' });
```

使用 JSX 后，工厂类方法已不太需要：JSX 提供了一种更简单、自然的方式来创建 ReactElement 实例。关于工厂方法这里不再赘述。

2.3.4 React的内部更新机制

在前面的例子中，我们创建了一个组件，并将其渲染到浏览器 DOM 中，其中 React 还做了哪些工作呢？这里主要概略介绍一下 React 内部的渐进式更新机制，读者只需了解其原理即可。具体的技术细节有兴趣的读者可以参看下面的附加阅读。

通常 React 并不直接操作浏览器 DOM，而是操作内部的虚拟 DOM。当数据发生改变时，React 先在虚拟 DOM 中自动计算判断出局部对应变更的部分，最后只将变更的部分反应到真实的浏览器 DOM 中。我们知道，频繁操作 DOM 会导致页面反复重画，其代价是昂贵的，而 React 创造性地解决了这个问题，这也是 React 页面响应快速的原因所在。当然，如果每次更新都是整体虚拟 DOM 发生变化，那么 React 快速的优势就不复存在了，但这种整体发生变化是很难发生的，因为这种情况完全可以设计为两个不同的组件了。实际使用中绝大多数变更都是局部变化，这也是 React 技术得以出现的前提。

对于开发者来说，只需要面对虚拟 DOM 就可以了，每次变化我们都看作虚拟 DOM 重新渲染一遍，但最终只变化局部，这样的机制极大地简化了应用的编写。对比传统的 jQuery 组件，我们往往不得不面对复杂的浏览器 DOM，查找和遍历各种标签去谨慎计算和控制浏览器 DOM 哪些区域发生变更，这需要“精打细算”且耗费大量的时间和精力。而使用 React 我们只需关注虚拟 DOM 就可以了，从而极大地提升了开发效率，同时思维的关注点不再是控制，而是表现，这也是 React 革新之处。

附加阅读：React 渐进式更新技术介绍

好的理念还需要强大的算法来支撑。React 巧妙地使用试探法将 $O(n^3)$复杂度的问题转换成 $O(n)$复杂度的问题。

将一颗树形结构转换成另一颗树形结构是一个复杂的、值得深入研究的课题。传统最优算法的复杂度是 $O(n^3)$，n 是树中节点的总数。这个代价非常昂贵，对于 1000 个节点要依次执行上十亿次的比较，按照当前 CPU 每秒执行大约三十亿条指

令来计算，即使是最优的实现，也太不可能在一秒内计算出差异情况。

而 React 巧妙地引入启发式算法思路，使用试探法实现了一个非最优但高效的 $O(n)$算法，该算法基于两个假定：

（1）相同类的两个组件将会生成相似的树形结构，而不同类的两个组件将会生成不同的树形结构。

（2）可以为元素提供一个唯一的标识，确保该元素在不同的渲染过程中保持不变。

这两个假定使算法具有出乎意料的性能，下面列举不同场景下算法实现细节。

（一）树形结构比较

为了进行一次树结构的差异检查，首先需要能够检查两个节点的差异。有三种不同的情况需要处理。

1. 不同的节点类型

如果节点的类型不同，React 将会把它们当作两个不同的子树，直接移除之前的那棵子树，然后创建并插入第二棵子树即可。该方法也同样适用于 HTML 标签。这就避开树形结构大部分的检测，然后聚焦于近乎相同的部分，实现了快速又精确的差异检测逻辑。比如在两个连续的渲染过程中的相同位置都有一个<Header>元素，则很有可能会生成非常相似的 DOM 结构，这才值得进行尝试匹配。

2. DOM 节点

对两个 DOM 节点进行比较时，通过查看两者的属性，对比找出随时间变化了的属性。对于内联样式，React 使用的是由键值对组成的对象，从而更容易更新那些改变了的样式属性。当属性更新完毕后，React 逐层递归检测所有子级的属性。

3. 自定义组件

对于自定义组件，因为组件是有状态的，不可能每次状态发生变化调用 render 时都在 render 函数中重新创建新的一系列组件实例。React 首先假定状态变化前后

的同一自定义组件是相同的，然后将变化后组件的所有属性合并到原组件实例上，再在原组件实例上调用 componentWillReceiveProps()和 componentDidReceiveProps()。至此，原组件实例的数据已经更新了，随后它的 render()方法被调用，然后差异算法重新比较新的状态和上一次的状态。

（二）子级列表的差异比较

对于子级列表的更新，React 使用的方法很原始：同时遍历两个子级列表，当发现差异的时候，就生成一次 DOM 修改，例如，在末尾添加一个元素就直接对应生成一个插入操作。但如果在列表的头部插入元素则比较麻烦，如下面的例子 React 发现两个节点都是 span，因此会直接修改已有 span 的文本内容，然后在后面插入一个新的 span 节点。

```
    renderA: <div><span>first</span></div>
    renderB: <div><span>second</span><span>first</span></div>
    => [replaceAttribute textContent 'second'], [insertNode <span>first
</span>]
```

有很多的算法尝试找出变换一组元素的最小操作集合。Levenshtein distance 算法能够找出这个最小的操作集合，使用单一元素插入、删除和替换，复杂度为 $O(n^2)$。但即使使用 Levenshtein 算法，也不会检测出一个节点已经移到了另外一个位置去了，要实现这个检测算法，只会使算法更复杂。

为了解决这个棘手的问题，React 引入了一个可选的键（key）属性（参见下一节）。通过给子级组件设定唯一的键值，React 就能够检测出节点插入、移除和替换，并且借助哈希表使节点移动复杂度变为 $O(n)$。

```
    renderA: <div><span key="first">first</span></div>
    renderB: <div><span key="second">second</span><span key="first">
first</span></div>
    => [insertNode <span>second</span>]
```

在实际开发中，生成一个唯一的键值不是很困难，比如给组件模型添加一个新

的 ID 属性或者计算部分内容的哈希值来生成一个键值。值得注意的是，键值只需要在兄弟节点中唯一即可，而不需要是全局唯一的。

（三）权衡

这里介绍的同步更新算法只是一种实现细节，也还可以有别的实现，只要 React 能在每次操作中重新渲染整个应用，最终的结果是一样的就行。将来这个启发式算法还会继续优化使得算法在常规的应用场景更加快速。

在当前的算法实现中，如果只能检测到某个组件已经从它的子级列表中移除，但是又不能知道它是否已经移到了其他某个地方。当前算法将会重新渲染整个子树。

由于算法依赖于前面提到的两个假定，如果这两个假定都没能满足，React 性能将会大打折扣。理解了算法的细节，我们就能更好地针对 React 在特定场景下进行优化，比如通过设置 key 值等。

原始英文资料来源于 https://facebook.github.io/react/docs/reconciliation.html，为符合中文表述习惯，这里略有加工。

2.3.5 虚拟DOM的特殊属性

由于虚拟 DOM 的特殊性，React 还提供了一些真实浏览器 DOM 中不存在的属性。

- key：可选的唯一标识符。当组件在渲染过程中被打乱的时候，按照差异检测逻辑，有时候可能会发生组件先被销毁而后又被重新创建的情况，产生不必要的开销。通过给组件绑定一个唯一的 key 属性，可以避免这样的问题，参见上节的附加阅读。
- ref：用于访问组件对应的实际 DOM 元素。在某些情况下，可能会需要直接操作一个组件渲染后所对应的 DOM 标记，比如要调整 DOM 元素的绝对位

置，或者在大型的非 React 应用中使用 React 组件，或者在 React 中重复使用已有的代码库等。React 给出的解决方法是使用 ref 属性，关于 ref 属性的使用，本书后面章节有详细介绍。

- dangerouslySetInnerHTML：用来直接插入纯 HTML 文本字符串，避免 React 的自动转义，参见 2.5.5 节。

2.4 JSX语法

React 开发者的核心理念认为前端应该组件化，组件应该与关注点分离而不是与模板和展现逻辑分离。传统的前端组件化其思路都是基于模板技术，如 Ext、jQuery UI 等，结构化标记和生成结构化标记的代码是紧密关联的。此外，展现逻辑一般都很复杂，使用模板语言会使展现变得笨重。React 解决这个问题的方式就是直接通过 JavaScript 代码生成 HTML 和组件树，这样，就可以使用 JavaScript 丰富的表达力去构建 UI。但满屏的 React.createElement 出现的时候，代码的可读性、可维护性会变得极差。为了使这个过程变得更简单，React 创建了类似 XML 的语法去构建虚拟 DOM 树，即 JSX。

JSX 即 JavaScript XML，它使用 XML 标记的方式来创建虚拟 DOM 和声明组件。前面所举的例子，从本质上已经能完成所有的工作了。只是有一些开发效率问题，比如 JavaScript 代码与标签混写在一起、缺乏模板支持等。而使用 JSX，则可以有效地解决这些问题。JSX 的实质是使用一种更为“自然”的方式来创建 ReactElement，它具有以下优点：

- 可以使用熟悉的语法仿照 HTML 来定义虚拟 DOM。
- JavaScript 语义增强，提供更加语义化的标签支持复杂应用创建。
- 程序代码更加直观。

- 抽象 ReactElement 创建过程。
- 支持变量嵌入和一些模板特性。
- 便于代码模块化。
- 与 JavaScript 之间存在等价转换。

以前文所举的 Helloworld 为例，使用 JSX 语法描述如下：

```
<!DOCTYPE html>
<html>
  <body>
    <div id="reactContainer"> </div>

    <script src="js/react.js"></script>
    <script src="js/react-dom.js"></script>
    <script>
      var HelloComponent = React.createClass({
        render: function() {
          return <h1<Hello world</h1>;
        }
      });

      ReactDOM.render(
        <HelloComponent/>,
        document.getElementById('reactContainer')
      );
    </script>
  </body>

</html>
```

从上面的代码可以看出，JSX 具有更简洁、更强大的表现力，而且基本不需要学习就能上手。

需要说明的是，JSX 是基于 ECMAScript 的一种新特性，但并不是一种新语言，它只是定义带属性的虚拟 DOM 树的一种语法，具体运行时，还需要通过转译器将 JSX 转换为原生 JavaScript 代码再执行。

前面已经指出，JSX 并不是必需的。前面的章节没有使用 JSX 描述主要是为了更清楚地了解 React 的基本原理。但在实际开发中，强烈推荐使用 JSX，因为它更为简洁、直观、易懂，能减少出错，且我们更为熟悉。本书下面的章节将全部采用 JSX 语法。

2.4.1 JSX等价描述

JSX 语法的特点是将 HTML 语言直接混写嵌入 JavaScript 语言中，而不需要加任何引号。下面来看看对比。

不使用 JSX 的例子：

```
React.render(
    React.createElement('div', null,
        React.createElement('div', null,
            React.createElement('div', null, 'content')
        )
    ),
    document.getElementById('reactContainer')
);
```

使用 JSX 的例子：

```
React.render(
    <div>
        <div>
            <div>content</div>
        </div>
    </div>,
```

```
    document.getElementById('reactContainer')
);
```

JSX 语法转译器会识别嵌入 JavaScript 代码中的 HTML 标签，当遇到“<”标识符就会启动 JSX 转译过程，遇到“{”标识符就会当作 JavaScript 代码进行处理，元素的标签、属性和子元素都会被当作参数传给 React.createElement 函数。

使用 JSX 语法的代码：

```
var app = <HelloComp color="blue" />;
```

使用原生 JavaScript 的等价代码：

```
var app = React.createElement(HelloComp, {color:"blue"});
```

2.4.2 JSX转译工具Babel

在实际使用时，JSX 代码会首先被转译为等价的原生 JavaScript 代码，然后再执行转译后的代码。官方最新推荐的 JSX 转译工具是 Babel（后文还有详细介绍）。早期的版本使用其自带的 JSX 语法编译器 JSTransform 进行解析，现已不推荐使用。而 Babel 作为专门的 JavaScript 语法编译工具，提供了更全面、更强大的功能，比如支持 ES 6 等。

转译过程可以在开发阶段完成，也可以在运行时完成，为提高性能，一般在开发完毕后将 JSX 代码转译为原生 JavaScript 代码再发布。如果在运行时完成转译，则需要加入对 Babel 中负责 JSX 转译的模块 browser.js 的引用，并将使用 JSX 语法描述的 js 文件类型声明为 text/babel，如下所示：

```
<script src="js/browser.js" type="text/JavaScript"></script>
<script src="js/HelloComponent.js" type=”text/babel"></script>
```

JSX 的转换还是有较大性能代价的，线上的代码最好还是直接执行原生 js 代码，因此我们常常在离线环境下提前完成转译后再进行部署，这里就要使用到 babel 离线转译工具。Babel 工具具体使用方法参见本书 10.6 节。

2.4.3 JSX中的表达式

JSX 支持使用 JavaScript 求值表达式作为属性值，从而达到类似模板的效果。JSX 中的表达式用一对大括号{}包起来。求值表达式是要求有返回值的表达式，其原理类似于 JSP 中的<%=...%>。求值表达式本身与 JSX 没有直接关系，只是 JS 的特性。

求值表达式与语句有所不同，在编写 JSX 时，在{}中不能使用语句（如 if 语句、for 语句等），但可以把语句放在函数中，再在求值表达式中调用该函数。

尽管在{}中不能使用 if-else 语句，但可以采用三元操作表达式。如：

```
    var HelloMessage = React.createClass({
      render: function() {
        return <div>Hello {this.props.name? this.props.name : "World"}
</div>;
      }
    });
    ReactDOM.render(<HelloMessage name="xiaowang" />, document.body);
```

也可以使用二元运算符“||”来书写，如果左边的值为真，则直接返回左边的值，否则返回右边的值，与 if 语句的效果相同。

```
    var HelloMessage = React.createClass({
      render: function() {
          return <div>Hello {this.props.name || "World"}</div>;
      }
    });
    ReactDOM.render(<HelloMessage name="xiaowang" />, document.body);
```

当三元操作表达式不能满足需要时，可以在 JSX 标签外使用 if 语句来决定应该渲染哪个组件。如：

```
    var loginButton;
    if (loggedIn) {
```

```
    loginButton = <LogoutButton />;
  } else {
    loginButton = <LoginButton />;
  }

  return (
    <nav>
      <Home />
      {loginButton}
    </nav>
  )
```

也可以使用变量来书写：

```
var HelloMessage = React.createClass({
  getName : function() {
      if (this.props.name)
          return this.props.name;
      else
          return "world";
  }
  render: function() {
      var name = this.getName();
      return <div>Hello {name}</div>;
  }
});
ReactDOM.render(<HelloMessage name="xiaowang" />, document.body);
```

或者把变量去掉，直接在{}中调用函数:

```
  render: function() {
      return <div>Hello {this.getName()}</div>;
  }
```

如果{}中包含的变量是一个数组，则会自动展开数组的所有成员，代码如下：

```
var persons = [
  <h1>Hello xiaoLi<h1>,
  <h2>Hello xiaoWang<h2>
];

ReactDOM.render(
  <div>{persons}</div>,
  document.getElementById('reactContainer')
);
```

2.4.4 JSX中的注释

在 JSX 中使用注释与在 JavaScript 中使用注释一样，本质上它就是 JavaScript 的一部分。但需要注意：当注释作为独立子节点时需要用{}包起来。如下所示：

```
var content = (
  <Comp>
    {/* 独立的注释要用{}包起来 */}
    <User
      /* 组件内的
         多行
         注释 */
      name="xiaoWang" // 行尾注释
    />
  </Comp>
);
```

2.4.5 JSX展开属性

如果在设计阶段就能明确组件属性，那么写起来会很容易。例如 component 组件有两个动态的属性 foo 和 bar：

```
var component = <Component foo={x} bar={y} />;
```

但实际上，有些属性可能是后续添加的，不能一开始就确定，于是我们可能会写出的代码：

```
var component = <Component />;
component.props.foo = x; // 错误
component.props.bar = y; // 错误
```

这样的写法是错误的且无法达到效果。因为在 React 的设定中，初始化 props 后，props 是不可变的。而且，手动直接添加的属性 React 后续是没办法检查到该属性的类型错误的。

为解决在运行中增加属性的问题，React 引入了属性展开机制。如下面的例子：

```
var extendProps = {};
extendProps.foo = x;
extendProps.bar = y;
var component = <Component {... extendProps } attr=”attrValue”/>
```

当需要增加组件的属性时，定义一个展开属性对象，并通过{... extendProps}的方式引入，React 会自动将 extendProps 中的属性复制到组件的 props 属性中。注意：这里的“…”是属性展开的特殊标识，ES 6（参见本书后面章节）中也使用“…”作为解构赋值中剩余属性标识，但两者没有关系。

展开属性会由转译器转译为类似下面的代码片段，可以看出，展开属性实际上是对属性进行合并后再传递。

```
React.createElement(ExampleApplication, _extends({}, extendProps,
{ attr: "attrValue"}), document.getElementById('container'));
```

有时候需要覆盖扩展属性中的原始属性值，则可以这样写：

```
var props = { foo: 'default' };
var component = <Component {...props} foo={'override'} />;
```

2.5 React注意事项

2.5.1 ReactDOM.render的目标节点

ReactDOM.render 的目标节点应该是单一的 DOM 节点，但不应该是 body，虽然这样做也能运行。因为 body 中往往还包含更多的子标签，如 script 标签等，会受到渲染的影响。如果确实要在 body 中渲染，可以动态添加一个 div 子标签后再渲染到这个子标签中，如：

```
ReactDOM.render(
   React.createElement(HelloComponent,null),
   document.body.appendChild(document.createElement('div'))
);
```

2.5.2 组件名约定

为了便于 JSX 正确识别是 HTML 标签还是 React 自定义组件，约定 JSX 中出现的 HTML 标签名统一为小写，React 自定义组件名首字母统一为大写。

2.5.3 class属性和for属性替换

声明在 JSX 中的标签属性中 class 必须以 className 代替，因为 class 已经是 JavaScript 最新语法规范中的保留字，为避免编译器的“误解”，JSX 中规定使用 className 代替。如：

```
<div className="myclass"…>
```

JSX 中的<label>标签的属性 for 必须用 htmlFor 代替，原因同上。如：

```
<label htmlFor="my-for-text"…>
```

2.5.4 行内样式

在 React 中，行内样式 style 属性并不是以 CSS 字符串的形式出现，而是一个特定的带有驼峰命名风格的 JavaScript 样式对象。这样的设计是为了与 DOM 中 style 的 JavaScript 属性名保持一致，且有助于弥补 XSS 安全漏洞。在这个样式对象中，key 值是用驼峰形式表示的样式名，而其对应的值则是样式值，通常是一个字符串。对于特定于浏览器的特殊属性，其浏览器前缀首字母除 ms 小写之外，其他字母都应该大写。示例如下：

```
var divStyle = {
  color: 'white',
  backgroundImage: 'url(' + imgUrl + ')',
  WebkitTransition: 'all', //针对特定浏览器的样式，首字母“W”要大写
  msTransition: 'all' //“ms”是针对微软浏览器的前缀，唯一一个首字母需要小写的
};

React.render(React.createElement(‘div’, {style:divStyle}, mountNode);
```

或直接写为：

```
React.render(<div style={{
color: 'white',
   backgroundImage: 'url(' + imgUrl + ')',
   WebkitTransition: 'all',
   msTransition: 'all'
}}></div>, mountNode);
```

注意：

此处的{{}}并无特殊含义，外层的{}是嵌入到 JSX 中 JavaScript 的识别符号，里层的{}是 JavaScript 中对象的声明方式。

2.5.5 自定义HTML属性

如果在编写 React 过程中针对 HTML 标签使用了自定义属性，正常情况下 React 是不会渲染的。如果要使用自定义属性，就需要给属性名加上 data-或 aria-前缀，如：

```
React.render(
    <div data-dd='xxx' aria-dd='xxx'>content</div>,
    document.body
);
```

需要提醒的是：HTML 5 中规定自定义属性 data-dd 中的 dd 都是小写字母，如果有大写会出错。如果确实需要对小写字母进行分隔，可以考虑使用 data-firstpart-secondpart 的方式。

对于自定义 React 组件，则没有上述限制，可以定义任意属性。

2.5.6 HTML转义

有时我们需要展示从后台取到的含有标签的富文本数据，如：

```
var content='<span>文本内容</span>';
React.render(
    <div>{content}</div>,
    document.body
);
```

结果页面直接输出内容：

```
<span>文本内容</span>
```

出现这种现象是因为 React 会默认进行 HTML 的转义，以避免 XSS 攻击，如果不需要转义，则可以使用 dangerouslySetInnerHTML 属性，使用方法如下：

```
var content='<span>文本内容</span>';
React.render(
   <div dangerouslySetInnerHTML={{__html: content}}></div>,
   document.body
);
```

3

React组件

从前面的章节已经知道 React 组件很简明，可以把它看作带有 props 属性集合和 state 状态集合并且构造出一个虚拟 DOM 结构的对象。其核心是 render 函数，该函数主要负责该组件的虚拟 DOM 构建。props 属性集合保存组件的初始属性数据；state 状态集合保存组件的状态数据；render 函数的主要职责是根据 state 状态，结合 props 属性，进行虚拟 DOM 的构建。React 的本质特点也在这里体现：render 只需要考虑根据状态生成对应的虚拟 DOM，其他所有的工作均由 React 自动完成，包括对变化的响应、重新渲染到浏览器等。所有的变化均由状态的变更引发，状态的变更通过调用组件实例的 setState 函数完成。

3.1 组件主要成员

3.1.1 state成员

组件总是需要与用户互动的。React 的一大创新，就是将界面组件看成一个状态机，用户界面拥有不同状态并根据状态进行渲染输出，用户界面和数据始终保持一致。于是开发者的主要工作就变成了定义 state，并根据不同的 state 渲染对应的用户界面。

通知 React 组件数据发生变化的方法是调用成员函数 setState(data, callback)。这个函数会合并 data 到 this.state，并重新渲染组件。渲染完成后，调用可选的 callback 回调。大部分情况下不需要提供 callback，因为 React 会负责把界面更新到最新状态。

下面我们来看一个小例子，完整示例参考所附实例包中 example-form 文件夹下 index.html 文件。该例包含一个文本框和一个按钮，单击按钮可以改变文本框的编辑状态：禁止编辑或允许编辑。通过这个例子来理解 React 的状态机制。先看代码：

```
var TextBoxComponent = React.createClass({
  getInitialState: function() {
    return {enable: false};
  },
  handleClick: function(event){
    this.setState({enable: !this.state.enable})
  },
  render: function(){
    return (
      <p>
        <input type="text" disabled={this.state.enable}/>
```

```
        <button onClick={this.handleClick}>改变 textbox 状态</button>
      </p>
    );
  }
});

ReactDOM.render(
  <TextBoxComponent/>,
  document.getElementById("reactContainer")
);
```

运行效果如下：

分析这个过程，组件最初有一个初始状态，通过调用方法 getInitialState 获得，这个方法在组件初始化的时候执行，返回一个对象或者 null。getInitialState 返回的对象会自动合并到 this.state 上，可以通过“this.state.属性名”的方式来访问属性值。这里将 enable 这个值和 input 的 disabled 绑定，当要修改这个属性值时，要使用 setState 方法。我们声明 handleClick 方法，来绑定到 button 上面，实现改变 state.enable 的值。当用户单击组件导致状态变化时，this.setState 方法修改状态值，每次修改以后，this.render 会被自动调用，从而再次渲染组件。

在这个示例中，在文本框中输入任意文本，再禁止文本框，可以发现文本框中原来输入的文本并没有消失，说明 React 并没有重新渲染 input，只是更改了 input 的属性值。这正是 React 差异对比、局部刷新的特性体现。

注意：

（1）getInitialState 函数必须有返回值，可以是 null、false、一个对象。

（2）访问 state 数据的方法是“this.state.属性名”。

（3）变量用{}包裹，不需要再加双引号。

3.1.2 props成员

props 是组件固有属性的集合，其数据由外部传入，一般在整个组件的生命周期中都是只读的，React 的 API 设计也决定了这一点。属性的初始值通常由 React.createElement 函数或者 JSX 中标签的属性值进行传递，并合并到组件实例对象的 this.props 中。事实上，组件从外界接收静态信息的主要渠道就是 props 属性。如：

```
var HelloBox = React.createClass({
  render: function() {
    return <div>{'Hello' + this.props. myattr }</div>;
  }
});

React.render(<HelloBox myattr="world"/>, mountNode);
```

或：

```
React.render(React.createElement(HelloBox,{myattr:'world'}),
mountNode);
```

此外，props 中也会包含一些由 React 自动填充的数据，比如 this.props.children 数组中会包含本组件实例的所有子组件，由 React 自动填充。如果需要，还可以通过配置实现 this.props.context 跨级包含上级组件的数据等。

我们不应该修改 props 中的值。事实上，如果要在组件外更改也是一件很麻烦的事：需要找到组件实例，这没有必要。在组件设计前期，就要有意识地分清组件哪些属性应该放在 props 集合，哪些属性应该放在 state 集合。通常，相对固定的、组件内只读的、应由父组件传递进来的属性适合放在 props 集合中，如组件的类名、颜色、字体、事件响应回调函数等。

props 与 state 的区别是：props 不能被其所在的组件修改，从父组件传进来的属性不会在组件内部更改；state 只能在所在组件内部更改，或在外部调用 setState 函数对状态进行间接修改。

3.1.3 render成员函数

对于组件来说，render 成员函数是必需的。render 函数的主要流程是检测 this.props 和 this.state，再返回一个单一组件实例，也可以返回 null 或 false 表明不需要渲染任何东西，对应的 React 渲染一个<noscript>标签来处理。当返回 null 或 false 的时候，this.getDOMNode()将返回 null。

render 函数应该是纯粹的，也就是说，在 render 函数内不应修改组件 state，不读写 DOM 信息，也不与浏览器交互，比如，调用 setTimeout 就是一种与浏览器的交互。如果确实需要和浏览器交互，应在 componentDidMount()中或者其他生命周期方法中进行设置。

注意：

React 组件只能渲染单个根节点。如果想要返回多个节点，它们必须被包含在同一个父级节点里。

从 render()中返回的内容并不是实际渲染出来的子组件实例，而仅仅是子组件层级树实例在特定时间的一个描述，React 会根据这个描述进行差异分析，再生成实际的组件实例。

3.2 组件的生命周期

一个组件完整的生命周期包含实例化阶段、活动阶段、销毁阶段三个阶段。每个阶段又由相应的方法管理。过程中涉及几个主要的动作术语：mounting 表示正在挂接虚拟 DOM 到真实 DOM，updating 表示正在被重新渲染，unmounting 表示正在将虚拟 DOM 移出真实 DOM。

3.2.1 实例化阶段

每个组件实例通过调用 React.createElement 方法完成实例化，ES 6 中则采用类继承的方式定义组件类（参看本书后面的章节，这里略过）。该方法会依据情况，分别调用该实例的以下方法。

1．getInitialState

用于对本组件实例的 state 进行初始化。对于每一组件实例，该方法会被调用且仅调用一次。通常，该方法会填充 state 初始数据，赋予组件实例的默认值。

2．componentWillMount

在初始化渲染执行之前立刻调用，该方法只执行一次。如果在这个方法内调用 setState，render()将会感知到更新后的 state。

3．render

该方法用于创建虚拟 DOM，表示组件的输出。对于一个组件类来讲，该方法是必须实现的。该方法通常从 props 中读取属性，从 state 中读取数据，并根据读取到的数据生成对应的虚拟 DOM。render 方法有以下几个要求：

- 只能读取 props 和 state，不能更改其中的值。
- 只能返回一个 ReactElement 元素或者 null、false。

render 方法返回的虚拟 DOM 被用于与浏览器 DOM 进行对比，并决定最终要更新的浏览器 DOM 内容。

4．componentDidMount

在 render 方法被调用，浏览器 DOM 被渲染之后，该方法被调用。此时的浏览器 DOM 已经被创建，且可通过调用 this.getDOMNode 获得对应的浏览器 DOM 元素，这是与 componentWillMount 最大的不同之处。该方法可用于对浏览器 DOM 做

进一步修改或其他操作。

通常，我们在这个方法里进行 state 数据的填充，比如进行 AJAX 请求。

如果要和其他 JavaScript 框架等集成，如 jQuery，应该在该成员函数中执行这些操作。

3.2.2 活动阶段

在这个阶段，组件的一系列初始化过程已经结束，对应的浏览器 DOM 也已经生成，主要关注的是对用户事件（如单击鼠标）的响应，对事件的响应通常会引发 state 的改变，进而导致组件的界面被刷新。

1．componentWillReceiveProps

父组件中的相关参数通常经由子组件的 props 进行传递。父组件 state 的变更也会引发子组件中对应 props 值的改变，此时子组件的 componentWillReceiveProps 函数将会被调用，这就为子组件的更新提供了机会。

componentWillReceiveProps 函数在组件接收到新的 props 的时候被调用。在初始化渲染的时候，该方法不会被调用。此函数是在 prop 传入之后渲染之前更新 state 的最佳时机。旧的 props 可以通过 this.props 获取到。在该函数中调用 this.setState() 不会引起第二次渲染。

```
componentWillReceiveProps: function(nextProps) {
  this.setState({
    likesIncreasing: nextProps.likeCount > this.props.likeCount
  });
}
```

注意：

对于 state，没有相似的方法：componentWillReceiveState。因为将要传进来的

prop 可能会引起 state 改变，反之则不然。如果需要在 state 改变的时候执行一些操作，应使用 componentWillUpdate。

2．shouldComponentUpdate

通常 React 会自动根据组件的虚拟 DOM 来决定浏览器 DOM 是否需要变更，但这个比对分析过程也是有代价的。shouldComponentUpdate 方法可以手动控制组件是否需要更新。如果该方法返回 false，则组件的 render、componentWillUpdate、componentDidUpdate 调用均会被忽略。

如果能确定新的 props 和 state 不会导致组件更新，则可直接返回 false。

```
shouldComponentUpdate: function(nextProps, nextState) {
  return false;
}
```

如果 shouldComponentUpdate 返回 false，则 render()将不会执行，直到下一次 state 改变。另外，componentWillUpdate 和 componentDidUpdate 也不会被调用。

默认情况下，shouldComponentUpdate 总会返回 true，即总是假定 state 在改变并通过对比逻辑进行更新。如果组件运行中 props、state 保持不变的，且在 render() 中只是读取 props 和 state 的值，这个时候覆盖 shouldComponentUpdate 方法，可以提高性能。

该方法可用于对性能进行优化，但大多数场景下没必要使用它，因为使用它有时也会带来一些副作用，如发生一些诡异的错误等。如果确实遇到了性能瓶颈，尤其是有几十个甚至上百个组件的时候，可以考虑小心地使用 shouldComponentUpdate 提升应用的性能，但需要仔细确认在带来明显的性能提升的同时并不会引发逻辑上的混乱，应尽量谨慎使用。

3．componentWillUpdate

在接收到新的 props 或者 state 之前立刻调用。在初始化渲染的时候该方法不会

被调用。使用该方法做一些更新之前的准备工作。该函数也应该谨慎使用。

注意：

不能在该方法中使用 this.setState()。如果需要更新 state 来响应某个 prop 的改变，请使用 componentWillReceiveProps。

4．componentDidUpdate

在组件的更新已经同步到 DOM 中之后立刻被调用。该方法不会在初始化渲染的时候被调用。使用该方法可以在组件更新之后操作 DOM 元素。该函数也应谨慎使用。

3.2.3 销毁阶段

componentWillUnmount

在组件从 DOM 中移除的时候立刻被调用。在该方法中执行任何必要的清理，比如无效的定时器，或者清除在 componentDidMount 中创建的 DOM 元素。

3.3 组件事件响应

React 在构建虚拟 DOM 的同时，还构建了自己的事件系统，这个事件系统与真实浏览器所遵循的 W3C 规范保持一致。具体地说，React 的所有事件对象和 W3C 规范保持一致，并且所有事件（包括提交事件）的行为都完整地遵循 W3C 规范，包括冒泡过程等。这使得事件在不同浏览器上有一致的表现。

稍微有些变化的是 onChange 事件，它在某些地方会和现有的浏览器表现不一致。事实上这种改变是 React 的一种改良，可以使我们对概念的理解更清晰。

React 的事件系统和浏览器事件系统相比，主要增加了两个特性：事件代理和事件自动绑定，这两个特性极大地方便了开发人员的工作。

3.3.1 事件代理

与浏览器事件处理方式不同的是，React 并未将事件处理函数与对应的 DOM 节点直接关联起来，而是在顶层使用一个全局的事件监听器监听所有事件。React 会在内部维护一个映射表记录事件与组件事件处理函数的对应关系，当某个事件发生时，React 根据这个内部映射表将事件分派给指定的事件处理函数。当映射表中没有事件处理函数时，React 不做任何操作。当一个组件安装或卸载时，相应的事件处理函数会自动被添加到事件监听器的内部映射表中或从表中删除。

3.3.2 事件自动绑定

在 JavaScript 中创建回调函数时，一般需要将方法绑定到特定实例，以保证回调函数中 this 值的正确性。在 React 中，每个事件处理回调函数都会自动绑定到组件实例（除非用 ES 6 语法）。React 通过缓存绑定的方法实现 CPU 和内存性能优化，同时也降低了开发人员的代码量。

注意，事件的回调函数被绑定在 React 组件上，而不是原始的元素上，即事件的回调函数中 this 所指的是组件实例而不是 DOM 元素。React 通过一个称为 autobinding 的过程自动将回调函数绑定到当前的组件实例上。

3.3.3 合成事件

与浏览器事件处理稍有不同的是，React 中事件处理程序所接收的事件参数是被称为“合成事件（SyntheticEvent）”的实例。合成事件是浏览器原生事件跨浏览器的封装，并与浏览器原生事件有着同样的接口，如 stopPropagation()、preventDefault()接口等，而且这些接口是跨浏览器兼容的。

如果由于某些特殊原因需要使用浏览器原生事件，可以通过合成事件的nativeEvent 属性获取。每个合成事件对象都有以下通用属性：

- boolean bubbles。
- boolean cancelable。
- DOMEventTarget currentTarget。
- boolean defaultPrevented。
- Number eventPhase。
- boolean isTrusted。
- DOMEvent nativeEvent。
- void preventDefault()。
- void stopPropagation()。
- DOMEventTarget target。
- Date timeStamp。
- String type。

注意：

早期版本的 React 通过在事件处理程序过程中返回 false 停止事件传播，现已不用。取而代之的是根据需要手动调用 e.stopPropagation()或 e.preventDefault()。

React 支持的事件如下表所示。

事件类型	事件名称	事件对象的专有属性（类型及变量）
剪贴板事件	onCopy onCut onPaste	DOMDataTransfer clipboardData

续表

事件类型	事件名称	事件对象的专有属性(类型及变量)
键盘事件	onKeyDown onKeyPress onKeyUp	boolean altKey Number charCode boolean ctrlKey function getModifierState(key) String key Number keyCode String locale Number location boolean metaKey boolean repeat boolean shiftKey Number which
焦点事件	onFocus onBlur	DOMEventTarget relatedTarget
表单事件	onChange onInput onSubmit	
鼠标事件	onClick onContextMenu onDoubleClick onDrag onDragEnd onDragEnter onDragExit onDragLeave onDragOver onDragStart onDrop onMouseDown onMouseEnter onMouseLeave	boolean altKey Number button Number buttons Number client*X* Number client*Y* boolean ctrlKey function getModifierState(key) boolean metaKey Number page*X* Number page*Y* DOMEventTarget relatedTarget Number screen*X* Number screen*Y* boolean shiftKey

续表

事件类型	事件名称	事件对象的专有属性(类型及变量)
触控事件	onTouchCancel onTouchEnd onTouchMove onTouchStart	boolean altKey DOMTouchList changedTouches boolean ctrlKey function getModifierState(key) boolean metaKey boolean shiftKey DOMTouchList targetTouches DOMTouchList touches
用户界面事件	onScroll	Number detail DOMAbstractView view
滚轮事件	onWheel	Number deltaMode Number delta*X* Number delta*Y* Number delta*Z*

3.4 props 属性验证

有时候我们希望对外界父级组件传递进来的属性数据进行限定,比如希望 name 属性不能缺少、onClick 属性必须是函数类型等，这对确保组件被正确使用非常有意义。为此 React 引入了 propTypes 机制。React.PropTypes 提供各种验证器 (validator) 来验证传入数据的有效性。当向 props 传入无效数据时，React 会在 JavaScript 控制台抛出警告。下面用例子来说明各类验证器的使用方法：

```
React.createClass({
  propTypes: {
    // 限定属性为 JavaScript 基本类型。默认情况下，这些属性是没有限制的
    optionalArray: React.PropTypes.array,
```

```
        optionalBool: React.PropTypes.bool,
        optionalFunc: React.PropTypes.func,
        optionalNumber: React.PropTypes.number,
        optionalObject: React.PropTypes.object,
        optionalString: React.PropTypes.string,

        // 限定该属性为所有可以被渲染对象,
        //包括数字、字符串、DOM 元素或包含这些类型的数组
        optionalNode: React.PropTypes.node,

        // 限定该属性为 React 元素
        optionalElement: React.PropTypes.element,

        // 限定该属性为 Message 类的实例
        optionalMessage: React.PropTypes.instanceOf(Message),

        // 限定该属性只接受指定值中的某一项
        optionalEnum: React.PropTypes.oneOf(['News', 'Photos']),

        // 限定该属性为指定多个对象类型中的一个
        optionalUnion: React.PropTypes.oneOfType([
          React.PropTypes.string,
          React.PropTypes.number,
          React.PropTypes.instanceOf(Message)
        ]),

        // 限定该属性为特定类型组成的数组
        optionalArrayOf: React.PropTypes.arrayOf(React.PropTypes.number),

        // 限定该属性为特定类型的对象
        optionalObjectOf: React.PropTypes.objectOf(React.PropTypes.
number),
```

```
    // 限定该属性为特定的复合对象
    optionalObjectWithShape: React.PropTypes.shape({
      color: React.PropTypes.string,
      fontSize: React.PropTypes.number
    }),

    // 限定该属性为函数类型且不可缺少
    requiredFunc: React.PropTypes.func.isRequired,

    // 限定该属性为不可空的任意类型
    requiredAny: React.PropTypes.any.isRequired,

    // 自定义验证器的示例。如果验证失败需要返回一个 Error 对象，
    // 不要直接使用“console.warn”或抛异常
    customProp: function(props, propName, componentName) {
      if (!/matchme/.test(props[propName])) {
        return new Error('Validation failed!');
      }
    }
  },
});
```

注意：

为了性能考虑，React 只在开发环境下验证 propTypes，运行环境下 propTypes 不起作用。

3.5 组件的其他成员

以下属性或方法均是可选的。

1．displayName

string displayName

displayName 字符串用于输出调试信息。JSX 会自动设置该属性。

2．getDefaultProps 函数

object getDefaultProps()

在组件类创建的时候调用一次，然后返回值被缓存下来。如果父组件没有指定 props 中的某个键，则此处返回的对象中的相应属性将合并到 this.props（使用 in 检测属性）。

该方法在任何实例创建之前调用，因此不能依赖于 this.props。另外，getDefaultProps() 返回的任何复杂对象将在实例间共享，而不是每个实例拥有一份复制件。

3．mixins

array mixins

mixins 数组用于定义在多个组件之间共享的函数代码。在组件的 mixin 中定义的函数会自动合并，成为组件的成员函数。有时候，不同组件之间可能共用一些功能、共享部分代码，使用 mixins 就能很好地满足这种需要。

4．statics

object statics

statics 对象用来定义静态的方法，这些静态方法可以在组件类上调用。例如：

```
var MyComponent = React.createClass({
  statics: {
    staticMethod: function(foo) {
      return foo === 'bar';
```

```
    }
  },
  render: function() {
  }
});

MyComponent. staticMethod('bar');  // true
```

在这段代码里定义的 staticMethod 函数都是静态的，这意味着可以在任何组件实例创建之前调用它们，这些函数不能获取组件的 props 和 state。如果确实需要在静态方法中检查 props 的值，可以在调用处把 props 作为参数传入到静态方法。

3.6 关于 state 的几个设计原则

3.6.1 哪些组件应该有state

大部分组件的原始数据应该来源于 props。只有对用户输入、服务器请求、时间变化等需要作出响应并暂存中间状态时才需要使用 state。

组件应该尽可能地无状态化。这样能减少冗余，同时使程序运作过程更明晰。

常见模式是创建多个只负责渲染数据的无状态（stateless）组件，在它们的上层创建一个有状态（stateful）组件，并把它的状态通过 props 传给子级。这个有状态的组件封装了所有用户的交互逻辑，而这些无状态组件则负责声明式地渲染数据。

3.6.2 哪些数据应该放入state中

state 应该包括可能被组件的事件处理器改变并触发用户界面更新的数据。实际场景中这种数据一般都很小且能被 JSON 序列化。当创建一个状态化的组件时，想

象一下表示它的状态最少需要哪些数据，并只把这些数据存入 this.state。在 render() 里再根据 state 来计算需要的其他数据。以这种方式思考和开发程序一般都是正确的，因为如果在 state 里添加冗余数据或计算所得数据，就需要经常手动保持数据同步，也就不能让 React 来协助进行处理。

3.6.3 哪些数据不应该放入state中

this.state 应该只包含能描述用户界面状态的最小数据集。因此，它不应该包含以下几种数据。

- 计算所得数据：没有必要把计算所得数据放在 state 中，把计算过程都放到 render()里更容易保证用户界面和数据的一致性。例如，在 state 里有一个数组（ items ），我们要渲染输出数组的长度值，直接在 render()里使用 this.state.items.length 即可，没有必要预先计算出它的长度并把长度值变量放到 state 里。
- React 组件：在 state 中不应该包含 React 组件，而只应该在 render()里创建。
- 基于 props 的重复数据：尽可能使用 props 来作为唯一数据来源。有时候可以把 props 保存到 state，比如在渲染时需要知道它以前的 props 值，以进行对比。

4

React顶级API

自 React 0.14 版本后，React 分解为 React、ReactDOM、ReactDOMServer 三个模块，分别对应 React、ReactDOM、ReactDOMServer 三个命名空间。

4.1 React命名空间

引用 React 库之后 React 是一个全局命名空间，包含了若干函数，这些函数基本上都是我们经常要用到的，熟练使用这些函数是开发 React 的基础。

1．React.createClass

```
ReactClass createClass(object specification)
```

用来定义组件类是最常用的函数。specification 参数描述组件类的成员函数，其中必须包含 render()函数，返回一个虚拟 DOM 的层级结构。组件类 ReactClass 代表一种逻辑封装，不同于标准原型类，其不使用 new 来实例化，而是调用 createElement 函数来间接实例化，最终生成的是 ReactComponent 对象。

2．React.Component

```
class Component
```

使用 ES 6 规范编写 React 组件时，通常采用继承类的方式定义组件类，而继承的基类就是 React.Component。使用继承类方式定义组件类的方式与普通 createClass 的方式起到的效果是相同的。

3．React.createElement

```
ReactElement createElement(
  string/ReactClass type,
  [object props],
  [children ...]
)
```

根据指定的组件类型或 HTML 标签名创建并返回一个代表组件实例的 ReactElement。

4．React.cloneElement

```
ReactElement cloneElement(
  ReactElement element,
  [object props],
  [children ...]
)
```

克隆所指定的组件并返回一个新组件。新组件中的属性值是原组件与 props 参数合并的结果。新组件中的 children 也是原组件与 children 参数合并的结果，如果

有重复，则以新的 children 参数为准覆盖原 children。这个函数在需要对子组件进一步进行加工时很有用。

5．React.createFactory

```
factoryFunction createFactory(
  string/ReactClass type
)
```

返回一个能生成指定类型 ReactElements 的工厂函数。这里的 type 参数可以是一个 HTML 标签名的字符串，或者是 ReactClass。有了 JSX 之后，这个函数就用得很少了。

6．React.isValidElement

```
boolean isValidElement(* object)
```

判断对象是否是一个 ReactElement。

4.2 ReactDOM命名空间

1．ReactDOM.render

```
ReactComponent render(
  ReactElement element,
  DOMElement container,
  [function callback]
)
```

渲染一个 ReactElement 到真实浏览器 DOM 中，并挂接到由 container 参数指定的 DOM 元素下，返回的是该组件实例的引用。

如果 ReactElement 之前已经被渲染到 container 中，该函数将会跟踪前后

ReactElement 中发生变化的部分，并在渲染到真实浏览器中的时候仅改变需要改变的 DOM 节点，以更新界面。

如果提供了可选的回调函数，则该函数将会在组件渲染或者更新之后调用。

2．ReactDOM.unmountComponentAtNode

```
boolean unmountComponentAtNode(DOMElement container)
```

从真实浏览器 DOM 中移除已经挂载的 React 组件，并清除相应的事件处理器和 state。如果在 container 内没有挂载任何组件，则什么都不做。如果有组件移除成功，则返回 true；如果没有组件被移除，则返回 false。

3．ReactDOM.findDOMNode

```
DOMElement findDOMNode(ReactComponent component)
```

如果组件已经被渲染到真实浏览器 DOM 中，调用这个函数可以获得与 component 组件相对应的真实 DOM 元素。如果还没有渲染，该函数会报错。这个函数常用于需要读取 DOM 元素信息，如表单值或元素尺寸等，也用于与普通 JavaScript 代码交互时。

一般没有必要直接使用 findDOMNode 函数，因为 React 通过 ref 属性机制可以直接获得真实 DOM 元素。

findDOMNode 函数要慎用，因为我们始终要关心组件是否已经被渲染。只要不是集成旧有代码所需，我们通常都可以通过良好的设计避开直接访问真实 DOM 这种方式。同时，这种方式造成了对真实浏览器 DOM 的依赖，也增加了以后向移动端迁移的开销。

此外，findDOMNode 函数也不能用于无状态组件。

4.3 ReactDOMServer命名空间

ReactDOMServer 命名空间位于 react-dom/server 模块，主要用于服务器渲染功能。因服务端渲染功能要求后台必须是 Node.js 环境，所以用途不广，本书也不再具体展开，仅进行简单列举。

1．ReactDOMServer.renderToString

```
string renderToString(ReactElement element)
```

用于把组件渲染成原始的 HTML 字符串，可以在服务器端预先生成 HTML 文本片段，然后将这个片段发送给客户端，这样可以获得更快的页面加载速度，并且有利于搜索引擎抓取页面。

如果在某个节点上调用 React.render()，且该节点已经有了服务器渲染标记，React 将自动管理该节点，只做事件绑定。

2．ReactDOMServer.renderToStaticMarkup

```
string renderToStaticMarkup(ReactElement element)
```

和 renderToString 类似，但是不创建额外的 DOM 属性，如 data-react-id 等，因为这些属性仅在 React 内部使用。比如用 React 做一个简单的静态页面生成器，丢掉额外的属性能够节省一些带宽。

4.4 children工具函数

组件的 this.props.children 包含该组件的所有子级组件实例，为方便查找、遍历子组件，React 提供了 React.Children 系列工具函数，专门用来处理 this.props.children 这个封闭的数据结构。

1．React.Children.map()

```
object React.Children.map(object children, function fn [, object context])
```

在包含于 children 参数中的每一个组件实例上调用 fn()函数，此函数中的 this 由参数 context 指定。每个 fn()都应有返回值，且所有的返回值会依次合并为一个对象作为最终的返回值。如果 children 是一个内嵌的对象或者数组，它将被遍历：不会传入容器对象到 fn()中。如果 children 参数是 null 或 undefined，那么返回 null 或 undefined，而不是一个空对象。

2．React.Children.forEach()

```
React.Children.forEach(object children, function fn [, object context])
```

类似于 React.Children.map()，但是不返回对象。

3．React.Children.count()

```
number React.Children.count(object children)
```

返回 children 中包含的组件总数，与传递给 map()或 forEach()的回调函数的调用次数一致。

4．React.Children.only()

```
object React.Children.only(object children)
```

返回 children 中仅有的子组件。如果子组件不存在或不唯一则会抛出异常。

5．React.Children.toArray()

```
array React.Children.toArray(object children)
```

返回由各子元素组成的数组，常用于在渲染事件中操作子元素集合，如重新排序或分割子元素等场合。

5

React表单

前端与用户发生交互的是表单组件，包括<input>、<textarea>、<option>等。表单组件不同于其他组件，因为它们会缓存用户的录入信息和录入状态，由此也带来了 React 对表单元素处理的一些特殊性。事实上，利用 React 机制，可以使表单组件的数据处理更加便利。

5.1 表单元素

表单组件支持几个受用户交互影响的属性：

- value，用于<input>、<textarea>组件。
- checked，用于类型为 checkbox 或者 radio 的<input>组件。

- selected，用于<option>组件。

在 HTML 中<textarea>的值通过子节点设置，在 React 中则应该使用 value 来设置。注意，for 要写成 htmlFor。提示信息现在多使用 input 的 placeholder 属性替代。

5.2 事件响应

表单组件可以通过设置 onChange()回调函数来监听组件变化。当用户的交互行为导致以下变化时，onChange()被执行并通过浏览器做出响应：

- <input>或<textarea>的 value 发生变化。
- <input>的 checked 状态改变。
- <option>的 selected 状态改变。

和所有 DOM 事件一样，所有的 HTML 原生组件都支持 onChange 属性，而且可以用来监听冒泡 change 事件。对于<input>和<textarea>，原生 DOM 内置的 onInput 事件应当用 onChange 替代。Checkbox 和 Radio 类型的 input 则稍有不同：当用户改变 Checkbox 和 Radio 时，React 用 click 事件代替 change 事件。大多数情况下，这种行为与我们的预期相同，但在调用 preventDefault 时要注意，即使 checked 被触发，preventDefault 也会阻止浏览器更新 input。这就是 React 在事件处理上与原生 DOM 不一样的地方。

若有多个元素要应用事件处理函数，常规的方法是编写多个 onChange 事件。但这样做会导致代码冗余，维护时会比较困难。更好的做法是只写一个事件处理函数且能处理多个事件，即复用。复用有 bind 复用和 name 复用两种方式。

5.2.1 bind复用

bind 方法为事件响应函数增加一个参数，事件响应函数通过该参数识别事件源。如下面代码所示，重点关注 handleChange 和 onChange 处。这种方式书写简单，性能也较好，推荐使用这种方式。JavaScript 的 bind()机制不在本书的讲解范围内，读者自行了解即可。

```
    var MyForm = React.createClass({
        getInitialState:function(){
            return {
                username:' ',
                gender:'男',
                checked:true
            };
        },
        handleChange:function(name, event){
            var newState={};
            newState[name]=name=="checked"?event.target.checked:event.
target.value;
            this.setState(newState);
        },
        submitHandler:function (e) {
            e.preventDefault();
            var is = this.state.checked ? "是" : "不是";
            var gender = this.state.gender == "man" ? "帅锅" : "美女";
            alert(this.state.username + is + gender + ".");
        },
        render:function () {
            return <form onSubmit={this.submitHandler}>
                <label htmlFor="username">请输入您的姓名：</label>
                <input type="text" name="username" onChange={this.Handle
Change.bind(this,"username")} value={this.state.username} id="username"/>
```

```
                <br/>
                <label htmlFor="checkbox">是或否：</label>
                <input type="checkbox" value="是否" checked={this.state.
checked} onChange={this.handleChange.bind(this,"checked")} name= "checked"
id="checkbox"/>
                <br/>
                <label htmlFor="username">请选择：</label>
                <select  name="gender"  onChange={this.handleChange.bind
(this,"gender")} value={this.state.gender}>
                    <option value="man">帅锅</option>
                    <option value="woman">美女</option>
                </select>
                <br/>
                <button type="submit">提交</button>
            </form>
        }
    });

    ReactDOM.render(
      <MyForm/>,
      document.getElementById('reactContainer')
    );
```

5.2.2 name复用

name 复用方式直接读取表单的属性值，比 bind 写法少一个参数（React 中事件响应函数会自动绑定 this）。其原理是在所有的标签中设置统一的 name 属性，并将这个属性值对应为 state 属性，在事件响应函数中通过读取表单的 name 值获得 state 属性，从 event.target.value 获取用户输入的值（check 控件稍有不同），要求所有相关的标签（包括 input 标签）都要统一设置 name 属性。

实例代码：

```
    var MyForm = React.createClass({
       getInitialState:function(){
          return {
             username:'',
             gender:'man',
             checked:true
          };
       },
       handleChange:function(event){
          var newState={};
    newState[event.target.name]=
    event.target.name=="checked"？event.target.checked：event.target.
value;
          this.setState(newState);
       },
       submitHandler:function (e) {
          e.preventDefault();
          var is = this.state.checked ? "是" : "不是";
          var gender = this.state.gender == "man" ? "帅锅" : "美女";
          alert(this.state.username + is + gender + ".");
       },
       render:function () {
          return <form onSubmit={this.submitHandler}>
             <label htmlFor="username">请输入您的姓名：</label>
             <input   type="text"   name="username"   onChange={this.
handleChange} value={this.state.username}/>
             <br/>
             <label htmlFor="checkbox">是或否：</label>
             <input  type="checkbox"  name="checked"  onChange={this.
handleChange} checked={this.state.checked}/>
```

```
                <br/>
                <label htmlFor="username">请选择：</label>
                <select name="gender" onChange={this.handleChange} value=
{this.state. gender}>
                    <option value="man">帅锅</option>
                    <option value="woman">美女</option>
                </select>
                <br/>
                <button type="submit">提交</button>
            </form>
        }
    });

    ReactDOM.render(
      <MyForm/>,
      document.getElementById('reactContainer')
    );
```

5.3 可控组件与不可控组件

在 React 中，input 标签的情况稍有不同：input 标签本身就有自己的状态缓存。由于 React 组件有 state 状态缓存，所以到底依托哪方缓存就决定了组件的性质，也决定了我们获取 input 标签中用户输入内容的方式——是直接访问 input 元素还是访问 React 组件的 state？这两种策略就决定了组件是否可控。将 input 中的 value 绑定到 state 的 React 组件就是可控组件，反之则是不可控组件。

5.3.1 可控组件

在 render()函数中设置了 value 的<input>是一个功能受限的组件，渲染出来的 HTML 元素始终保持 value 属性的值，即使用户输入也不会改变。如：

```
    render: function() {
       return <input type="text" value="Hello"/>;
     }
```

上面的代码将渲染出一个值始终为 Hello 的 input 元素，这是由 React 的渲染策略决定的。如果需要响应更新用户输入的值，就需要使用 onChange 事件，并将 value 绑定到 state 中。

```
     getInitialState: function() {
       return {value: 'Hello!'};
     },
     handleChange: function(event) {
       this.setState({value: event.target.value});
     },
     render: function() {
       var value = this.state.value;
       return <input type="text" value={value} onChange={this.handle
Change} />;
     }
```

可以看到，一个可控组件并不保持自己的原始状态；组件的呈现完全基于 React 的 state 属性。在情况允许的条件下，应优先选择可控组件方式。

可控组件具有以下优点：

- 符合 React 的单向数据流特性，即从 state 流向 render 输出的结果。
- 数据存储在 state 中，便于访问和处理。

5.3.2 不可控组件

不将 value 绑定到 state 上的<input>组件是一个不可控组件。这样组件中的数据和 state 中的数据并不对应，也就无法从 state 中获取用户输入的信息，所以组件的数据不可控。

```
render: function() {
   return <input type="text" />;
}
```

上面的代码将渲染出一个空值的输入框，用户输入将立即反映到元素上。和受限元素一样，使用 onChange 事件可以监听值的变化。

如果想给组件设置一个非空的默认初始值，可以使用 defaultValue 属性。数据在这里并没有存储在 state 中，而是包含在 input 元素中。

```
render: function() {
   return <input type="text" defaultValue="Hello!" />;
}
```

如果要获得 input 中的 value，需先拿到其 DOM 节点，然后获取其 value 值：

```
var inputValue = React.findDOMNode(this.refs.input).value
```

上面的代码渲染出来的元素和受限组件一样，也有一个初始值，但这个值用户可以改变并会反映到界面上。

同样，<input type="checkbox">和<input type="radio">支持 defaultChecked 属性，<select>支持设置 defaultValue。defaultValue 和 defaultChecked 属性只能在初始的 render()函数中使用，如果要在随后的 render()函数中更新 value 值，则应使用可控组件。看下面的例子：

```
var UncontrolComponent = React.createClass({
  onFormSubmit: function(e) {
    e.preventDefault();
    var helloRef = ReactDOM.findDOMNode(this.refs.helloUnke).value;
```

```
    alert(helloUnke);
  },
  render: function(){
    return (
      <form onSubmit={this.onFormSubmit}>
        <input ref="helloRef" type="text" value="Hello!"/>
        <button type="submit">speak</button>
      </form>
    );
  }
});

ReactDOM.render(
  <UncontrolComponent/>,
  document.getElementById('reactContainer')
);
```

6

React复合组件

React 是基于组件化的开发，组件化的目的自然是为了复用。基于 React 的应用界面是由一个个组件嵌套组合形成的。复合的组件要进一步分解为多个子组件，组件还要遵循职责单一原则。组件外在体现的是可复用性，内在体现的则是封装性。通过封装明确边界，将组件内在实现与外在接口区分开，屏蔽了组件的内部实现细节。那么，剩下的主要就是组件之间如何嵌套组合、如何传递信息以及如何共享信息等问题了。

6.1 组件嵌套

父组件要包含子组件有两种方式：一种是在父组件的 render 函数中直接渲染子组件，这种方式假定子组件的类型和属性均是已知的，这也是通用的方式。

```
var Student = React.createClass({
  render: function() {
return (
     <li>{this.props.name}</li>
    );
  }
});

var StudentList = React.createClass({
  render: function() {
return (
     <ul>
       <Student name="张三"></Student>
       <Student name="李四"></Student>
     </ul>
    );
  }
});
ReactDOM.render(
  < StudentList />,
  document.getElementById('reactContainer')
);
```

然而，如果子组件类型和属性是设计时未知的或者是动态加入的，就需要用到 this.props.children 属性。React 会自动将所有子组件的实例填入 this.props.children 属性中。父组件可以根据这个属性与子组件进行各种操作。下面的例子实现了与上面的例子同样的效果，但子组件是在父组件 render 函数之外声明的。需要注意的是，组件的父子关系并不等同于标签的父子关系。

```
var StudentList = React.createClass({
  render: function() {
return (
```

```
        <ul>
        {this.props.children}
        </ul>
      );
    }
  });

  ReactDOM.render(
    < StudentList >
      <Student>张三</Student>
      <Student>李四</Student>
    </ StudentList >,
    document.getElementById('reactContainer')
  );
```

比较以上两种方式，我们到底应该用哪种呢？对于固定数量和属性的子组件，我们倾向于使用前者。而对于动态的、数量不固定的子组件我们更倾向于采用后者，这样的代码结构更清晰且易于控制。

对于后者，我们有时候还需要对子组件进行更进一步的控制，比如过滤其中的某些子组件或限制必须为某种类型的子组件，这就需要自行扩展。可以参看某些开源实现，如 react-bootstrap 的源代码。

6.2 组件参数传递

6.2.1 动态参数传递

通常情况下，子组件明确地知道从 props 中传入的属性，甚至要对传入的属性进行限定。但父组件向子组件传递参数时，有时参数名称是作为变量出现的，无法

预先明确下来，这就是动态的参数传递。动态参数传递的方法是使用展开属性的方法，直接传递一个属性对象，但需要加入特殊的标识符“...”，如下：

```
    return <Component {...this.props} more="values" />;
```

有时把所有属性都传下去是冗长且不安全的。这时可以使用 ES 6 规范解构赋值中的剩余属性特性，把未知属性批量提取出来。下面的示例将 this.props 中除了 checked 属性之外的其他属性复制到 other 变量中再传递给组件，但这里的“...”表示剩余属性，与上例中出现的“...”展开属性是不一样的。

```
    var { checked, ...other } = this.props;
    return <Component {...other} attrMore="values" />;
```

如果上面的 other 对象中也含有 attrMore 属性，则会被覆盖。这里的顺序很重要，把{...other}放到 JSX props 前面会使它不被覆盖。

6.2.2 使用Underscore来传递

如果不使用 JSX，可以使用一些库来实现相同的效果。Underscore 提供_.omit()来过滤属性，_.extend()复制属性到新的对象。不过这种方法已经不常用。

```
    var FancyCheckbox = React.createClass({
    render: function() {
    var checked = this.props.checked;
    var other = _.omit(this.props, 'checked');
    var fancyClass = checked ? 'FancyChecked' : 'FancyUnchecked';
    return ( React.DOM.div(_.extend({}, other, { className: fancy
Class })) );
    }
    });
```

6.2.3 使用Context 来传递

在数据通过组件树逐层传递时，有时某些属性会从上层组件手动逐层传递到最底层的组件，比如 A 组件为了给包含在 B 组件中的 C 组件传递一个 prop ，需要把参数在组件中传递两次才能最终将 A 组件中的 prop 传递给 C 组件 。这样的传递既烦琐又无聊，而且使 B 组件在中间参与了与自己无关的逻辑。针对这个问题，我们可以使用 Context 特性，其能实现组件树上的数据越级传递。

Context 是 React 0.14 版本新增的一个高级的、实验性的机制，将来的 API 细节可能会有一些更改，不推荐频繁使用。如果确实需要，应尽量保持在小范围内使用，并且避免直接使用 Context 的 API，以便于以后的升级。

官方文档的示例代码如下：

```
    var Button = React.createClass({
      render: function() {
        return (
          <button style={{background: this.props.color}}>
            {this.props.children}
          </button>
        );
      }
    });

    var Message = React.createClass({
      render: function() {
        return (
          <div>
            {this.props.text}  <Button  color={this.props.color}> 删 除
</Button>
          </div>
        );
      }
```

```
});

var MessageList = React.createClass({
  render: function() {
    var color = "purple";
    var children = this.props.messages.map(function(message) {
      return <Message text={message.text} color={color} />;
    });
    return <div>{children}</div>;
  }
});
```

将上面的例子改为使用 Context 机制进行数据传递，重点关注加粗部分和相应注释。

```
var Button = React.createClass({
  // 使用 context 时下级组件必须指定 context 的数据类型
  contextTypes: {
    color: React.PropTypes.string
  },
  render: function() {
    return (
      //使用 this.context 来获取 context 数据
      <button style={{background: this.context.color}}>
        {this.props.children}
      </button>
    );
  }
});

var Message = React.createClass({
  render: function() {
    return (
```

```
        <div>
          {this.props.text} <Button>删除</Button>
        </div>
      );
    }
  });

  var MessageList = React.createClass({
    //使用 context 时上级组件必须要声明的属性
    childContextTypes: {
      color: React.PropTypes.string
    },
    //使用 context 时上级组件必须要声明的回调函数
    getChildContext: function() {
      return {color: "purple"};
    },
    render: function() {
      var children = this.props.messages.map(function(message) {
        return <Message text={message.text} />;
      });
      return <div>{children}</div>;
    }
  });
```

在示例代码中，在下级组件 Button 中定义 contextTypes，并通过 this.context 访问数据，在上层组件 MessageList 通过添加 childContextTypes 和 getChildContext() 提供数据，React 自动向下传递数据，使得组件树中上层组件的任意层级的下级组件都能获得上层组件中的数据。Context 机制在 React 的路由管理框架 router 中也有应用。

6.3 组件间的通信

前面介绍的主要是针对父子组件或上下层级组件之间单向的参数传递机制。如果两个组件之间不具备这样的关系，则需要一些特殊机制。当然，这种跨组件之间的通信需求本身也有其特殊性。

6.3.1 事件回调机制

父级组件通过子级组件的 props 传递数据并控制子级组件的行为。如果子级组件要向父级组件传递信息，该怎么办呢？通常的办法是父级组件在 props 中增加一个回调函数，子级组件在恰当的时机调用这个回调函数通知父级组件，从而实现两者的通信，这需要子级组件预先约定好这个回调接口，并且父级组件遵从这个约定。

看下面的例子，这个例子在实例包中对应 chapter5/example-communication 文件夹：

```
var ChildComponent = React.createClass({
  handleClick: function(){
    if (this.props.onClick) {
      this.props.onClick();
    }
  },

  render: function(){
    return (
        <button onClick={this.handleClick}>单击我通知父组件</button>);
  }
});
```

```
var ParentComponent = React.createClass({
  onChildClicked: function() {
    alert('父级组件收到了子级组件的单击事件。');
  },
  render: function(){
    return (
      <div>父组件
        <ChildComponent onClick={this.onChildClicked}/>
      </div>
    );
  }
});

ReactDOM.render(
  <ParentComponent/>,
  document.getElementById('reactContainer')
);
```

对于没有隶属关系的组件间的通信，也可以通过相似的事件机制来实现通信。在 componentDidMount()里订阅事件，在 componentWillUnmount()里退订，然后在事件回调里调用 setState()。前提是组件设计时就约定了相应的事件。

6.3.2 公开组件功能

另外一种不常用的实现通信的方法是子组件对外提供公开方法。公开的方法被调用后返回相应的数据。父组件通过 ref 属性获得子组件实例的引用。

以一个待办事项列表为例，单击该列表项后该项被删除。如果只剩下一个未完成的待办事项，则在控制台输出一段文本模拟一个动画过程：

```
var TodoItem = React.createClass({
```

```
    render: function() {
      return <div onClick={this.props.onClick}>{this.props.title}</div>;
    },
    //这是本组件公开的方法，由父组件通过 ref 属性获取实例并调用
    animate: function() {
      console.log(' %s 的 animate 函数被调用', this.props.title);
    }
  });
  var Todos = React.createClass({
    getInitialState: function() {
      return {items: ['Apple', 'Banana', 'Cranberry']};
    },
    handleClick: function(index) {
      var items = this.state.items.filter(function(item, i) {
        return index !== i;
      });
      this.setState({items: items}, function() {
        if (items.length === 1) {
          //此处调用子组件的 animate 函数
          this.refs.item0.animate();
        }
      }.bind(this));
    },
    render: function() {
      return (
        <div>
          {this.state.items.map(function(item, i) {
            var boundClick = this.handleClick.bind(this, i);
            return (
            <TodoItem onClick={boundClick} key={i} title={item} ref=
{'item' + i} />
            );
```

```
        }, this)}
      </div>
    );
  }
});
React.render(<Todos />, mountNode);
```

当然，要达到同样的效果，也可以采取不同的策略，比如给某个 TodoItem 赋予 isLastUnfinishedItem 属性，并由这个 TodoItem 本身进行动画控制等。这里不评价策略好坏，只是用来验证机制和思路。

6.3.3 mixins

组件是 React 里复用代码的最佳方式，但有时一些复杂的组件间也需要共用一些功能或具有一些共同的行为，如输出日志等，有时这也称为跨切面关注点。React 使用 mixins 机制解决这类问题。由于组件之间共享逻辑，所以也能通过这个逻辑实现组件之间的通信与协作。

举一个典型的例子，很多组件都会有定时更新界面的需求。用 setInterval()实现定时器的操作并不难，只是当不需要定时器时要及时清除定时器，以减小内存开销，尤其是大量的组件都包含定时器时。React 的组件生命周期方法可以告知我们组件创建或销毁的时机，但如果多个组件都需要定时更新机制时，我们为每个组件都实现一套定时器的创建和销毁机制显然是不可取的，最好是将这个机制共享并使其能为多个组件所用，这时可以使用 mixins 机制。下面使用 setInterval()来做一个简单的 mixin，并保证在组件销毁时清理定时器。

```
var SetIntervalMixin = {
  componentWillMount: function() {
    this.intervals = [];
  },
  setInterval: function() {
```

```
    this.intervals.push(setInterval.apply(null, arguments));
  },
  componentWillUnmount: function() {
    this.intervals.map(clearInterval);
  }
};

var TickTock = React.createClass({
  mixins: [SetIntervalMixin], // 引用 mixin
  getInitialState: function() {
    return {seconds: 0};
  },
  componentDidMount: function() {
    this.setInterval(this.tick, 1000); // 调用 mixin 中的方法
  },
  tick: function() {
    this.setState({seconds: this.state.seconds + 1});
  },
  render: function() {
    return (
      <p>
        React 已经运行了{this.state.seconds}秒.
      </p>
    );
  }
});

React.render(
  <TickTock />,
  document.getElementById('reactContainer')
);
```

简单地说，在 mixins 中定义的函数被“混”入组件实例中，多个组件定义相同的 mixins 则会使组件具有某些共同的行为。

SetIntervalMixin 中也定义了 componentWillMount 函数，在这种情况下，React 会优先执行 mixin 中的 componentWillMount。如果 mixins 中定义了多个 mixin，则按声明的顺序依次执行，最后执行组件本身的函数。

如果一个组件使用了多个 mixin，并且有多个 mixin 定义了同样的生命周期方法（如：多个 mixin 都需要在组件销毁时做资源清理操作），所有这些生命周期方法都会被执行到：首先按 mixin 引入顺序执行 mixin 里的方法，最后执行组件内定义的方法。

在最新的 React 中 mixins 机制已不再是优先提倡对象，我们大致了解其原理即可。

6.3.4 动态子级

一个 React 复合组件会包含多个子组件，如果在运行的过程中子组件的位置会发生变化（如在搜索结果中）或者有新的子组件插入到列表开头，React 会难以识别这些组件的变化关系，从而导致子组件删除后重建。如果要确保子组件在多个渲染阶段保持自己的特征和状态，就需要给每个子组件都赋予一个唯一标识 key。

```
render: function() {
  var results = this.props.results;
  return (
    <ol>
      {results.map(function(result) {
        return <li key={result.id}>{result.text}</li>;
      })}
    </ol>
  );
}
```

当遇到带有 key 的子组件时，React 会确保它们被重新排序而不是重建。值得注意的是，key 只对组件有效，对 HTML 元素不起任何作用。

6.4 高阶组件

React 现在已不建议推广使用 mixins 功能了，因为 React 提供了更好的选择——高阶组件（higher-order component）。mixins 虽然能实现跨组件的逻辑共享，但也有一些副作用，比如破坏了封装性原则等。而高阶组件以更好的方式实现了跨组件的逻辑共享，同时又具有很多优良的特性。现在很多模块都使用了高阶组件，著名的 react-redux 框架中的 connect 函数，以及 redux-form 中都有高阶组件的例子。高阶组件实质是对现有组件的动态包装函数，与普通函数稍有区别的是它接受组件类型作为输入参数，并输出经过包装后的新组件类型，这也是其被称为高阶的原因。

6.4.1 高阶组件概念

为理解高阶组件，我们先从需求入手。有的时候，我们需要对现有组件增加或修改功能，但是又不希望这种改动影响到现有的依赖于此组件的应用。此时就要对这个组件进行一个包装，生成一个新的组件。如下面的例子：

```
class NewComponent extends Component {
    //增加或修改的功能实现
    extendFunc(){
}
    render() {
        return <OldComponent {... this.props} />
    }
}
```

上面的例子已经完全能解决问题了。但是，如果有较多的组件时，我们是否要在每个组件中都增加 extendFunc 逻辑？这时我们应该考虑将组件类型作为变量，利用函数来封装：

```
function hoc(Comp) {
   return class NewComponent extends Component {
//增加或修改的功能实现
     extendFunc(){
}
     render() {
       return (
         <Comp {... this.props}/>
       )
     }
   }
 }
```

使用时直接调用函数就可在任意原组件的基础上生成包含新功能的新组件：

```
const NewComponent= hoc(Comp);
```

上面的例子利用 JavaScript 语言的函数和闭包特性，改变了已有组件的行为而完全不需要修改源代码，这就是高阶组件的工作方式。由此可以看出，高阶组件只是一种用法或一种架构，并没有增加新的内容。

从实际运行结果来看，组件树的嵌套虽然多了一层或多层，但是实际渲染出来的 DOM 结构并没有改变。使用多层高阶组件不会影响输出的 DOM 结构，对性能也没有影响。借助函数的逻辑表现力，高阶组件得到了越来越广泛的应用。

6.4.2 高阶组件应用：属性转换器

若我们想要的组件与现有组件在功能上相似，但组件的参数并不一致，就可以给现有组件加装一个属性转换适配器，使之满足我们的需求。下面的代码就实现了组件属性转换：

```
function transProps(transFn) {
    return function(Comp) {
        return class extends Component {
            render() {
                return <Comp {...transFn(this.props)}/>
            }
        }
    }
}
```

使用时直接调用函数：

```
const NewAdapter = transProps (transPropsFu)(OldComp);
```

此写法在函数的基础上又增加了一阶函数，以将 transPropsFu 属性转换逻辑与 OldComp 组件逻辑分离开，这样做关注点更明确。借助高阶组件，代码的结构、逻辑更清晰。

6.4.3 高阶组件应用：逻辑分离与封装

我们倾向于设计单一目标的组件，因为这样的组件易写易测。然而在实际项目中，我们却往往容易混淆这一点。以典型的包含异步数据请求的组件为例，假设我们需要异步加载学生列表的一个组件，普通的设计可能是这样的：

```
class StudentList extends Component {
  constructor(props) {
    super();
    this.state = {
      listData: []
    }
  }
  componentDidMount() {
    loadStudents()
```

```
        .then(data=>
          this.setState({listData: data.studentList})
        )
    }
    render() {
      return (
        <List list={this.state.listData} />
      )
    }
  }
```

这样的设计很普遍，看似也没什么问题。但我们再进一步思考一下：数据请求方式一定是 Fetch 方式吗，如果换成 jQuery 中的 ajax 函数呢？另外，若请求到的结果中列表数据不是学生列表而是课程列表呢，难道每种情况都设计一个新的组件吗？可是我们的本意只是想要一个能以列表的方式显示数据的组件。问题出在哪儿呢？究其原因，上面的代码将数据请求和数据展现两种逻辑混在一起，使组件无法适应更多的场景，从而限制了它的可重用性。两种逻辑应该分离，那如何实现逻辑分离？

使用高阶组件能很好地解决这个问题，将数据展现逻辑封装为一个组件，再通过高阶组件包装这个组件并附加数据请求逻辑，就能达到效果。

```
  function hocListWithData({dataLoader, getListFromResultData}) {
    return Comp=> {
      return class extends Component {
        constructor(props) {
          super();
          this.state = {
            resultData: undefined
          }
        }
        componentDidMount() {
```

```
          dataLoader()
            .then(data=> this.setState({resultData:data}))
        }
        render() {
          return (
            <Comp {... getListFromResultData (this.state.resultData)}
{...this.props}/>
          )
        }
      }
    }
```

使用时分别调用 hocListWithData()函数，并传入不同的参数，即可生成不同的组件类。高阶组件有时也可理解为一个更高层次组件的生成器。

```
    const StudentList = hocListWithData({
       dataLoader: loadStudents,
       getListFromResultData: result=> ({listData: result.studentList})
    })(List);

    const LessionList = hocListWithData({
       dataLoader: loadLessons,
       getListFromResultData: result=> ({listData: result.lessonList})
    })(List);
```

通过引入高阶组件，不仅大大减少了重复代码，还把交织在一起的请求逻辑和展示逻辑分离到不同的层次中进行封装，从而为独立的管理和测试提供了更好的支持。有趣的是，在这个例子的基础上，我们还可以进一步封装出更高阶的组件，以增加更多的逻辑，比如为上面的高阶组件增加一阶，以实现属性转换的功能等。

高阶组件也是函数式编程的生动示例，体现了函数式编程的强大之处。下面是 React-Redux 中的一段源代码，connectToStores 通过组件内嵌的方式，将 store 与组件连接起来，通过高阶组件生动地体现了设计思想。

```
function connectToStores(Component, stores, getStateFromStores) {
  const StoreConnection = React.createClass({
    getInitialState() {
      return getStateFromStores(this.props);
    },
    componentDidMount() {
      stores.forEach(store =>
        store.addChangeListener(this.handleStoresChanged)
      );
    },
    componentWillUnmount() {
      stores.forEach(store =>
        store.removeChangeListener(this.handleStoresChanged)
      );
    },
    handleStoresChanged() {
      if (this.isMounted()) {
        this.setState(getStateFromStores(this.props));
      }
    },
    render() {
      return <Component {...this.props} {...this.state} />;
    }
  });
  return StoreConnection;
};
```

7

React常用组件示例

为了更好地理解 React 技术并使其与实践相结合，本章给出了几个常用组件的设计思路和具体源代码，并对设计过程进行了分析。这些示例工程均位于实例包 chapter7/example-demo 文件夹下，读者可以运行示例查看具体效果。

7.1 按钮组件

按钮是 Web 应用开发过程中最常见的元素之一。按钮组件实际上基于两种标签实现：<A>标签和<button>标签，用户可以自行决定使用哪类标签，默认使用<button>标签。

为保持页面风格一致，开发人员一般需要预先定义适用于各种场景的按钮的样式类，如在 Bootstrap 中定义的 btn-default、btn-success、btn-danger、btn-warning 等

分别用于默认、操作成功、危险操作、警告的场景中。

在用 React 实现按钮组件时，可以从样式和事件两个方面考虑。样式应该允许使用者自行定义，并作为属性传入组件，若用户不使用该属性，组件需要提供默认样式。一般情况下，按钮仅响应鼠标单击事件，用户可将自定义的事件处理函数作为属性传入组件，事件触发时，该函数会被调用。

在本示例中，我们构造了一个按钮组件 ButtonComponent，该组件将返回一个 Bootstrap 3 中的默认按钮，并可接受用户自定义的单击事件处理方法。

在组件定义时，我们使用 propTypes 特性对传入组件的属性进行了类型约束，限定属性 clickHandler 为函数类型，当传入属性不是函数类型时，JavaScript 控制台将输出警告信息。

请注意，在 render 函数中，定义按钮单击事件时，并不是简单地将 clickHandler 属性指定为事件处理函数，而是使用语句 this.props.clickHandler.bind(this)。在这里，bind(this)会改变事件处理函数的执行环境，即将 clickEventHandler()函数中的 this 替换为 bind(this)中的 this（这里的 this 指向 ButtonComponent）。单击按钮时，JavaScript 控制台将输出当前的按钮组件对象，如果去掉 bind()函数，控制台将输出 null。

```
    var ButtonComponent = React.createClass({
      propTypes: {
        clickHandler: React.PropTypes.func
      },
       render: function() {
          var tagType = this.props.tag && 'button';
             if (tagType == 'button') {
                return (
                   <button className="btn btn-default"
                                onClick={this.props.clickHandler.
bind(this)}>
                      {this.props.children}
                   </button>
```

```
                );
            } else {
                return (
                    <a className="btn btn-default" href="javascript:
void(0);"
                                        onClick={this.props.click
Handler.bind(this)}>
                        {this.props.children}
                    </a>
                );
            }
        );
    }
});
function clickEventHandler(e) {
    console.log(this);
    console.log('捕获单击事件成功!');
}
ReactDOM.render(
        <ButtonComponent clickHandler={this.clickEventHandler}>提交
</ButtonComponent>,
    document.getElementById('button-component')
);
```

7.2 分页组件

分页组件一般配合表格使用，主要实现对表格型数据的分页控制，包括显示页面总数、当前页码等信息，提供数据刷新、页面切换（前后页跳转、首末页跳转、页间跳转）控制，并可配置页面数据行数等功能，如下图所示。

从组成上来看，分页组件由一组按钮、一个输入框、一个文件显示区域构成。

分页组件的数据来源应该包括对相关信息的描述，应该包含 pageSize、pageNum、pages、offset、total、col_name、direction 等数据。

分页组件内部需要响应这一组按钮的 click 事件和输入框的 onChange 事件。同时，分页组件的使用者需要传入 onFirst、onLast、onPrev、onNext、onRefresh、onChange 六个回调函数，分别对应首页、尾页、上一页、下一页、刷新五个按钮单击事件和每页数据条目数更改事件。组件接受一个输入对象属性 data，其具体结构为：

```
    {
    columns: [{key: String, label: String, type: String}, …],
    items: Array,
    paginate: { pageSize: Number, pageNum: Number, pages: Number, offset:
Number, total: Number, col_name: Number , direction: Number}
    }
```

为提高易用性，在当前页面已是第一页（最后一页）时，首页和上一页按钮（尾页和下一页按钮）将变为无效状态，即 disabled 属性为 true。

```
    var Foot = React.createClass({
        propTypes: {
            onFirst: React.PropTypes.func,
            onPrev: React.PropTypes.func,
            onNext: React.PropTypes.func,
            onLast: React.PropTypes.func,
            onRefresh: React.PropTypes.func,
            onChange: React.PropTypes.func,
            data: React.PropTypes.shape({
                columns: React.PropTypes.array,
                paginate: React.PropTypes.shape({
```

```
                pageSize: React.PropTypes.number,
                pageNum: React.PropTypes.number,
                pages: React.PropTypes.number
            })
        })
    },
    render: function () {
        return (
            <tfoot>
                <tr>
                    <td colSpan={this.props.data.columns.length}>
                        <div className="pull-left">
                            <Button text="<< 首页" onClick={this.props.
onFirst} disabled={this.props.data.paginate.pageNum === 1} />
                            <Button text="< 上一页" onClick={this.props.
onPrev} disabled={this.props.data.paginate.pageNum === 1} />
                            <Button text="下一页 >" onClick={this.props.
onNext} disabled={this.props.data.paginate.pageNum === this.props.data.
paginate.pages} />
                            <Button text="尾页 >>" onClick={this.props.
onLast} disabled={this.props.data.paginate.pageNum === this.props.data.
paginate.pages} />
                            <Button  text="刷 新 "  onClick={this.props.
onRefresh} disabled={false} />
                        </div>
                        <div className="pull-left" style={{marginLeft:
'30px'}}>
                            <span>每页</span>
                            <select    onChange={this.props.onChange}
value={this.props.data.paginate.pageSize}
                                    className="page-number-select"
name="pageSize">
```

```
                                <Option value="5" />
                                <Option value="10" />
                            </select>
                            <span>行</span>
                        </div>
                        <div className="pull-right">
                            <span className="footer-style">第{this.props.
data.paginate.pageNum}页(共{this.props.data.paginate.pages}页)</span>
                        </div>
                    </td>
                </tr>
            </tfoot>
        );
    }
});
```

这里 Foot 组件使用了 Button 和 Option 组件，与上一节的 Button 组件相比，这里的 Button 组件还需传入 disabled 属性，通过该属性可激活和禁用按钮。

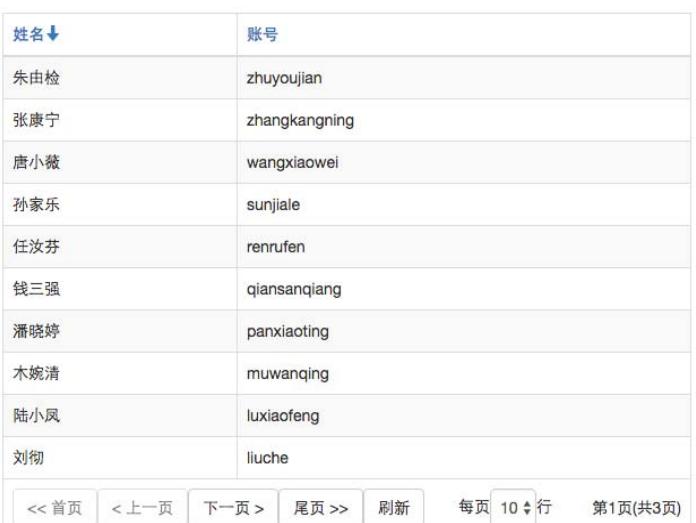
```
var Button = React.createClass({
    propTypes: {
        disabled: React.PropTypes.bool
    },
    render: function () {
        console.log("render button onClick=" + this.props.onClick);
        return (
            <button type="button" className="btn btn-default"
onClick={this.props.onClick} disabled={this.props.disabled}>
                {this.props.text}
            </button>
        );
    }
});
```

```
    var Option = React.createClass({
        render: function () {
            return (<option value={this.props.value}>{this.props.value}
</option>);
        }
    });
```

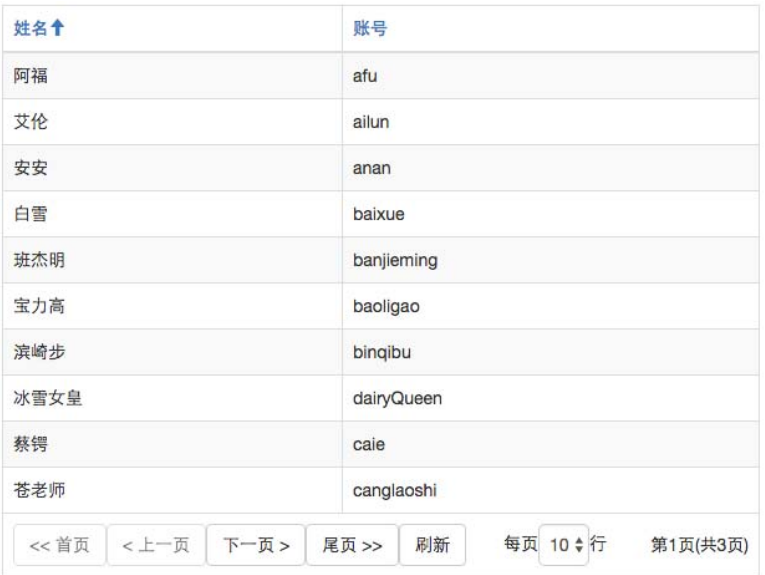
7.3 带分页的表格组件

开发 Web 应用时，经常需要进行扁平结构数据集的显示，这时就不得不用到表格控件。Sencha Ext、jQuery EasyUI、Kendo UI 等前端 JavaScript 框架都提供了表格控件，这些控件内置了排序、过滤、分页等功能，可极大地提高 Web 应用的开发效率。

本示例基于 React 设计实现了一个带排序、分页功能的表格组件。为简化设计，该组件仅完成了前端排序和分页，而服务器端排序、分页功能开发的主要工作是前后端参数对接及后端数据库查询操作，对前端组件依赖较少，这里不做考虑。运行效果如下：



对姓名列进行排序后效果如下：



姓名↑	账号
阿福	afu
艾伦	ailun
安安	anan
白雪	baixue
班杰明	banjieming
宝力高	baoligao
滨崎步	binqibu
冰雪女皇	dairyQueen
蔡锷	caie
苍老师	canglaoshi

表格组件 GridComponent 是表格和分页组件的结合体。GridComponent 组件由表格头组件 Head、表格体组件 Body 和分页组件 Foot 构成。组件的分页和排序功能分别由 loadData 方法和 sortDataFunc 方法实现。loadData 方法首先根据当前分页大小重新计算页码和分页数，随后计算当前页码对应的数据起止索引，取出相应的数据，最后将计算结果赋值到组件状态并最终反映到页面上。sortDataFunc 方法按照顺序和倒序声明了两个比较函数，可对数据的指定列进行顺序或倒序排序，方法通过 this.state.data.paginate.direction 指定顺序，通过 this.state.data.paginate.col_name 指定要排序的列。实现了 getFirst、getPrev、getNext、getLast、changeRowCount 方法，用以响应分页组件的首页、上一页、下一页、尾页按钮单击和每页行数更改事件；实现了 sortData 方法，响应表格头每列的鼠标单击事件并进行排序。

具体代码如下：

```
var GridComponent = React.createClass({
    sortDataFunc: function () {
        //按字符串正向排序
```

```
            function ascCompare(propertyName) {
                return function (obj1, obj2) {
                    var value1 = obj1[propertyName];
                    var value2 = obj2[propertyName];
                    if(!value1) return -1;
                    if(!value2) return 1;
                    if(typeof value1 == 'string')
                        return value1.localeCompare(value2);
                    return 0;
                }
            }
            //按字符串逆向排序
            function descCompare(propertyName) {
                return function (obj1, obj2) {
                    var value1 = obj1[propertyName];
                    var value2 = obj2[propertyName];
                    if(!value1) return 1;
                    if(!value2) return -1;
                    if(typeof value1 == 'string')
                        return -value1.localeCompare(value2);
                    return 0;
                }
            }
            if(this.state.data.paginate.direction =='asc')
                all_items.sort(ascCompare(this.state.data.paginate.
col_name));
            else
                all_items.sort(descCompare(this.state.data.paginate.
col_name));

        },
        loadData: function() {
```

```
            //分页大小是变化的，需要判断当前页面是否超出范围
            var  pages  =  Math.ceil(all_items.length/this.state.data.
paginate.pageSize);
            var pageNum = this.state.data.paginate.pageNum;
            if(pageNum > pages)pageNum = pages;

            //计算当前页包含的数据项的起止范围并取出
            var start = (pageNum-1) * this.state.data.paginate.pageSize
+ this.state.data.paginate.offset;
            var  endp  =  start  +  parseInt(this.state.data.paginate.
pageSize);
            var end = (endp > all_items.length) ? all_items.length : endp;
            var result = all_items.slice(start, end);

            //构造新的组件状态
            var data = {
               columns:  [{key:'userName',label:'姓名',type:'String'},
{key:'userAccount',label:'账号',type:'String'}],
               items: result,
               paginate: {
                  pageNum: pageNum,
                  pageSize: this.state.data.paginate.pageSize,
                  pages:  pages,
                  offset: 0,
                  total: all_items.length,
                  col_name: this.state.data.paginate.col_name,
                  direction: this.state.data.paginate.direction
               }
            };
            this.setState({paginate: data.paginate});
            this.setState({data: data});
         },
```

```
       getInitialState: function () {
           return {
               data: {
                   columns: [{key:'userName',label:'姓名',type: 'String'},
{key:'userAccount',label:'账号',type:'String'}],
                   items: [],
                   paginate: {
                       pageNum: 1,
                       pageSize: 10,
                       pages:  Math.ceil(all_items.length/10),
                       offset: 0,
                       total: all_items.length,
                       col_name: "userName",
                       direction: "asc"
                   }
               }
           };
       },

       componentDidMount: function () {
           this.sortDataFunc();
           this.loadData();
       },

       getFirst: function() {
           this.setState({paginate: $.extend(this.state.paginate, { pageNum:
1 })});
           this.loadData();
       },

       getPrev: function() {
```

```
            this.setState({paginate: $.extend(this.state.paginate,
{ pageNum: this.state.paginate.pageNum - 1 })});
            this.loadData();
        },

        getNext: function() {
            this.setState({paginate: $.extend(this.state.paginate,
{ pageNum: this.state.paginate.pageNum + 1 })});
            this.loadData();
        },

        getLast: function() {
            this.setState({paginate: $.extend(this.state.paginate,
{ pageNum: this.state.paginate.pages })});
            this.loadData();
        },

        changeRowCount: function(e) {
            var el = e.target;

            this.setState({paginate: $.extend(this.state.paginate,
{ pageSize: el.options[el.selectedIndex].value })});
            this.loadData();
        },

        sortData: function (e) {
            e.preventDefault();
            var el = e.currentTarget,
                col_name = el.getAttribute("data-column"),
                previousDirection = el.getAttribute("data-direction");
            var direction = (previousDirection === "desc") ? "asc" :
"desc";
```

```
            console.log(previousDirection,el);
            this.setState({paginate: $.extend(this.state.paginate,
{ col_name: col_name, direction: direction })});
            this.sortDataFunc();
            this.loadData();
        },

        render: function () {
            return (
                <table  className="table  table-striped  table-bordered
table-hover">
                    <Head data={this.state.data} onSort={this.sortData}
/>
                    <Body data={this.state.data} />
                    <Foot data={this.state.data} onFirst={this.getFirst}
onPrev={this.getPrev}   onNext={this.getNext}   onLast={this.getLast}
onChange={this.changeRowCount} onRefresh={this.loadData}/>
                </table>
            );
        }
    });
```

Head 组件用于展现表格每列的名称，包含多个 HeadCell 组件，每个 HeadCell 实现一个列的名称展现，并监听鼠标单击事件对相应的列进行排序。Head 组件根据当前排序列 this.props.data.paginate.col_name 计算 showArrow，并将其作为属性传递到 HeadCell 组件，HeadCell 组件以此判断某一列是否应该显示排序标志。HeadCell 组件根据 direction 属性决定排序箭头方向。

```
    var Head = React.createClass({
        render: function () {
            return (
                <thead>
```

```
                <tr>
                    {this.props.data.columns.map(function (column, i) {
                        return   <HeadCell   key={i}   column={column}
direction={this.props.data.paginate.direction} onSort={this.props. onSort}
showArrow={this.props.data.paginate.col_name == column.key}/>
                    }.bind(this))}
                </tr>
            </thead>
        );
    }
});
var HeadCell = React.createClass({
    propTypes: {
        showArrow: React.PropTypes.bool
    },
    render: function () {
        var arrow =  "glyphicon-arrow-up";
        if(this.props.direction === "desc" ){
            arrow =  "glyphicon-arrow-down";
        }
        var iDom = (this.props.showArrow)? (<i className={"glyphicon
"+ arrow}/>):null;
        return (
            <th>
                <a href="#" data-column={this.props.column.key} data-
direction={this.props.direction === "desc" ? "desc" : "asc"}
                   role="button"  tabIndex="0"  onClick={this.props.
onSort}>
                    {this.props.column.label}
                    {iDom}
                </a>
            </th>
```

```
            );
        }
    });
```

Body 组件用于真正显示数据内容，其结构较为直观，一个 Body 组件包含多个 Row 组件，一个 Row 组件则由多个 Cell 组件构成，Cell 组件用于展示单个数据项。这里，Cell 组件兼容数据、字符串和图片类型的数据项展示。

```
    var Body = React.createClass({
        render: function () {
            return (
                <tbody>
                    {this.props.data.items.map(function(item, i) {
                      return <Row key={i} item={item} columns={this.props.
data.columns} />
                    }.bind(this))}
                </tbody>
            );
        }
    });
    var Row = React.createClass({
        render: function () {
            return (
                <tr>
                    {this.props.columns.map(function (column, i) {
                      return <Cell key={i} column={column} value={this.
props.item[column.key]} />
                    }.bind(this))}
                </tr>
            );
        }
    });
```

```
    var Cell = React.createClass({
        renderCell: function(column, value) {
            switch (column.type) {
                case 'Number':
                    return value;
                    break;
                case 'String':
                    return value;
                    break;
                case 'Image':
                    return React.createElement('img', {src: value}, null);
                    break;
            }
        },

        render: function () {
            return (<td>{this.renderCell(this.props.column, this.props.
value)}</td>);
        }
    });
    ReactDOM.render(
        <GridComponent/>,
        document.getElementById("grid-component")
    );
```

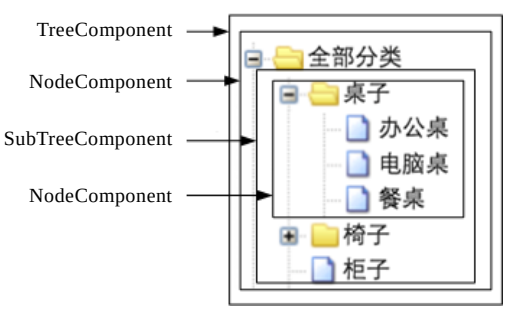
7.4 树形组件

在 Web 应用开发中，会碰到一些树形结构，如组织机构、类别目录等，需要使用树形控件才能展示数据的顺序和上下级连接关系。Sencha Ext、Kendo UI 等

JavaScript 框架也都提供了树形控件。另外，还有一款著名的由中国人开发的开源 jQuery 树形插件——zTree，实现了多级数据展现、鼠标事件响应、节点展开/收起等功能，受到广泛欢迎。

本示例参考 zTree 插件，基于 React 实现了一个具有多级数据展现、节点展开/收起、单击事件函数自定义的 React 树形组件 TreeComponent。TreeComponent 组件接受的数据结构为对象数组，每个对象包含 id、pId、name、open（可选）等属性，在 TreeComponent 组件的创建代码中已经对 nodeList 属性类型进行限定。

TreeComponent 组件根据 nodeList 属性构造了名为 nodeList 的状态，之所以增加该状态，一方面是因为用户可能会对树形结构的节点进行重命名、增、删或变更父子关系等编辑操作（本例未实现这些功能），另一方面是因为扁平的数组结构无法从父节点直接找到全部子节点，而本例构造的 nodeList 状态是多级的树状结构，父节点包含一个由其全部子节点组成的数组。



由于树形结构的级数可变，需要用递归的方式才能实现树形组件的定义。如上图所示，树形组件可被看作一组节点组件（NodeComponent）的集合，每个节点组件包含开合按钮、图标、名称及其子树组件（SubTreeComponent，仅在当前节点不是叶子节点时有子树组件），子树组件与树形组件类似，也由一组节点组件组成。当待处理节点全是叶子节点时，递归结束。

节点组件（NodeComponent）包含一个名为 open 的状态，组件根据 this.props.node.nodeList 是否为空来判断当前节点有无子节点，根据 open 状态的值来判断展开或折叠子节点。

```
    var TreeComponent = React.createClass({
        //限定各属性的类型
        propTypes:{
            treeId: React.PropTypes.string.isRequired,
            onClickFunc: React.PropTypes.func,

            //每个节点数据中必须包含 id 和 pId
            nodeList: React.PropTypes.arrayOf(
                React.PropTypes.shape({
                    id: React.PropTypes.oneOfType([
                        React.PropTypes.string.isRequired,
                        React.PropTypes.number.isRequired,
                    ]),
                    pId: React.PropTypes.oneOfType([
                        React.PropTypes.string.isRequired,
                        React.PropTypes.number.isRequired,
                    ]),
                    name: React.PropTypes.string.isRequired,
                    open: React.PropTypes.bool
                })
            )
        },

        getDefaultProps: function() {
            return {
                //onClickFunc 属性（鼠标单击节点事件的处理函数）可以不提供，此时
需给一个默认值
                onClickFunc: function () {}
            };
        },
        //根据传入的 nodeList 属性，构造一个树状的节点结构——nodeList 数组
        getInitialState:function () {
```

```
            var nodeList = [];
            for(var i=0;i<this.props.nodeList.length;i++){
                var found = false;
                for(var j=0;j<this.props.nodeList.length;j++){
                    if(i==j)continue;

                    //找到每个节点的父节点，并将其放入父节点的节点列表中
                    if(this.props.nodeList[i].pId == this.props.nodeList
[j].id){
                        found = true;
                        if(!this.props.nodeList[j]["nodeList"])
this.props.nodeList[j]["nodeList"]=[this.props.nodeList[i]];
                        else
this.props.nodeList[j]["nodeList"].push(this.props.nodeList[i]);
                    }
                }
                //找不到父节点的节点即为根节点，放入 nodeList 数组中；非根节点均
存在根节点的子节点列表中
                if(!found){
                    nodeList.push(this.props.nodeList[i]);
                }
            }
            //将 nodeList 数组作为组件状态返回
            return {nodeList:nodeList};
        },
        render: function () {
            return (
                //一个树形组件可被看作由多个节点组件的列表构成，每个节点组件包含开
合按钮、图标、名称及其子树组件（仅在当前节点不是叶子节点时有子树组件）
                <ul id={this.props.treeId} className="react_tree">
                    {this.state.nodeList.map(function (node, i) {
                        return <NodeComponent key={node.id + "_child_" + i}
```

```
node={node}    treeId={this.props.treeId}    onClickFunc={this.props.
onClickFunc}/>
                }.bind(this))}
            </ul>
        );
    }
});

var SubtreeComponent = React.createClass({
    render: function () {
        return (
            <ul id={this.props.subtreeId} className="line">
                {this.props.nodeList.map(function (node, i) {
                    return <NodeComponent key={node.id + "_child_" + i}
node={node}    treeId={this.props.treeId}    onClickFunc={this.props.
onClickFunc}/>
                }.bind(this))}
            </ul>
        );
    }
});

var NodeComponent = React.createClass({
    getInitialState:function () {
        return {open:this.props.node.open == true};
    },
    openOrClose: function () {
        this.setState({open:!this.state.open});
    },
    //响应鼠标单击节点事件，将调用用户自定义的 onClickFunc 方法
    onNodeClick: function (event) {
        var treeId = this.props.treeId,
```

```
            treeNode = this.props.node;
         //除 event 外，还将 treeId，treeNode 两个参数传入 onClickFunc 方法
         this.props.onClickFunc.call(this, event, treeId, treeNode);
      },
      render: function () {
         var openClass = "center_close", icoClass = "ico_close";
         if(!this.props.node.nodeList || this.props.node.nodeList.
length == 0){
            openClass = "center_docu";
            icoClass = "ico_docu";
         }
         else if(this.state.open){
            openClass = "center_open";
            icoClass = "ico_open";
         }
         var idPrefix = this.props.treeId + "_" + this.props.node.id;
         return (
            <li id={idPrefix}>
               <span id={idPrefix + "_switch"} className={"button
switch " + openClass} onClick={this.openOrClose}></span>
               <a id={idPrefix + "_a"} onDoubleClick={this.
openOrClose} onClick={this.onNodeClick}>
                  <span id={idPrefix + "_ico"} className={"button "
+ icoClass}></span>
                  <span id={idPrefix + "_span"}>{this.props.node.
name}</span>
               </a>
               {(openClass == "center_open")?
                  <SubtreeComponent subtreeId={idPrefix + "_ul"}
treeId={this.props.treeId}       nodeList={this.props.node.nodeList}
onClickFunc={this.props.onClickFunc}/>: ""
               }
            </li>
```

```
            );
        }
    });

    function onClickFunc(event, treeId, treeNode) {
        console.log(event, treeId, treeNode);
    }

    ReactDOM.render(
        <TreeComponent treeId={"myTree"} nodeList={treeNodes} onClickFunc=
{onClickFunc}/>,
        document.getElementById("tree-component")
    );
```

7.5 模态对话框组件

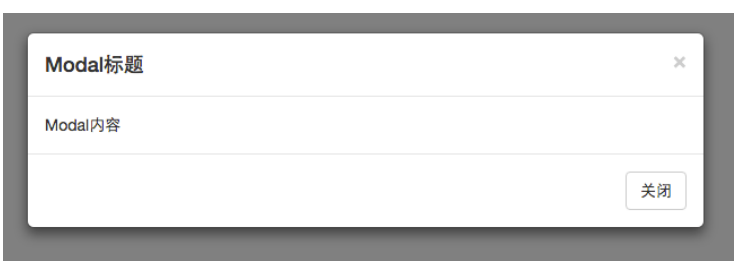
模态对话框（Modal Dialog Box）是一种特殊的页面元素，当模态对话框打开后，用户必须先对该对话框进行响应（如关闭），才能继续进行主界面的操作。其实现原理是弹出对话框时，在当前页面先覆盖一个遮罩层，再在之上显示模态框元素。目前，基于 jQuery 的模态框种类繁多，Bootstrap 也提供了自己的模态框。

本示例所实现的模态框仿照 react-bootstrap 的 Modal 组件设计而成，因而需要用到层叠样式表（Cascading Style Sheets，CSS）文件。本示例的 ModalComponent 组件实现了模态框的打开和关闭功能，可通过单击 Header 中的“×”符号、关闭按钮或模态框外部区域关闭模态框。

本示例的 ModalComponent 组件渲染时会在页面的 Body 元素内部末尾添加一个 标签，弹出框和屏蔽层都放在该 标签内部，关闭模态框时删除该 标签即可。屏蔽层的 z-index 值较大，能够屏蔽除弹出框外的其他所有元素，从而

有效阻止用户对模态框下面的页面元素进行单击和其他交互操作。

ModalComponent 组件的代码与常见的组件定义不同，最大的区别是该组件的 render 方法始终返回 null，不执行任何动作，而是在 componentDidMount 和 componentDidUpdate 方法中（第一次渲染完成后和每次组件更新传递给真实 DOM 后）调用一个名为_renderOverlay()的函数来完成真正的渲染。_renderOverlay()函数在真实 DOM 的 body 元素内部末尾插入一个空的 标签，然后调用 ReactDOM 的 unstable_renderSubtreeIntoContainer 方法将模态框的内容渲染到这个空的 标签内。当模态框关闭或组件即将卸载（componentWillUnmount）时，通过调用 ReactDOM 的 unmountComponentAtNode 方法卸载 标签中的 React 组件，然后删除 标签。其运行效果如下图所示。



MyModal 组件展示了一个示例模态框组件的创建方式，该组件拥有一个名为 showModal 的状态，并提供了两个方法（onClose 和 onOpen）对该状态进行操作，实现模态框的打开和关闭功能。用户可以采用示例提供的 Header、Body 和 Footer 对模态框内容进行规范布局，也可按照实际情况在 ModalComponent 标签内添加自己的模态框内容。

```
var ModalComponent = React.createClass({
    //限定各属性的类型
    propTypes:{
        show: React.PropTypes.string,
        onHide: React.PropTypes.func.isRequired,
        ariaLabelledby: React.PropTypes.string,
    },
```

```
        handleDialogClick: function(e) {
            //当鼠标单击的目标是事件绑定元素的子节点时，忽略该单击事件
            if (e.target !== e.currentTarget) {
                return;
            }

            this.props.onHide();
        },
        //组件加载后，进行模态框的渲染
        componentDidMount: function() {
            this._renderOverlay();
        },
        //组件更新后，进行模态框的渲染
        componentDidUpdate: function() {
            this._renderOverlay();
        },
        //组件卸载前，分别从虚拟 DOM 和真实 DOM 中移除模态框叠层
        componentWillUnmount: function() {
            this._unrenderOverlay();
            this._unmountOverlayTarget();
        },
        //在真实 DOM 的 body 元素内部的末尾插入模态框叠层
        _mountOverlayTarget: function() {
            if (!this._overlayTarget) {
                this._overlayTarget = document.createElement('div');
                this._portalContainerNode = document.getElements ByTagName
("body")[0];
                this._portalContainerNode.appendChild(this.
_overlayTarget);
            }
        },
        //在真实 DOM 中移除模态框叠层
```

```
        _unmountOverlayTarget: function() {
            if (this._overlayTarget) {

this._portalContainerNode.removeChild(this._overlayTarget);
                this._overlayTarget = null;
            }
            this._portalContainerNode = null;
        },
        //在虚拟 DOM 中插入模态框叠层及内容
        _renderOverlay: function() {
            var children = this.props.show?(
                <div>
                    <div  className="modal-backdrop  fade  in"  onClick=
{this.props.onHide}></div>
                    <div className="modal in fade" role="dialog" aria-
labelledby={this.props.ariaLabelledby} onClick={this.handleDialogClick}
                         style={{display: 'block', paddingLeft: 0}}>
                        <div className="modal-dialog">
                            <div className="modal-content">
                                {this.props.children}
                            </div>
                        </div>
                    </div>
                </div>
            ):(
                null
            );

            if (children !== null) {
                this._mountOverlayTarget();
                this._overlayInstance = ReactDOM.unstable_renderSubtree
IntoContainer(
```

```
                this, children, this._overlayTarget
            );
        } else {
            //隐藏模态框时，从虚拟 DOM 和真实 DOM 中移除模态框叠层
            this._unrenderOverlay();
            this._unmountOverlayTarget();
        }
    },

    //在虚拟 DOM 中移除模态框叠层
    _unrenderOverlay: function() {
        if (this._overlayTarget) {
            ReactDOM.unmountComponentAtNode(this._overlayTarget);
            this._overlayInstance = null;
        }
    },

    //渲染函数不执行任何动作
    render: function() {
        return null;
    }
});
//模态框顶部组件，含标题和关闭按钮
ModalComponent.Header = React.createClass({
    propTypes:{
        closeButton: React.PropTypes.bool,
        onHide: React.PropTypes.func.isRequired,
    },
    render: function () {
        const {
            'aria-label': label,
            closeButton,
```

```
            children,
            onHide
        } = this.props;

        return (
            <div className="modal-header">
                {closeButton &&
                <button
                    type="button"
                    className="close"
                    aria-label={this.props.label}
                    onClick={onHide}
                >
                    <span aria-hidden="true">
                      &times;
                    </span>
                </button>
                }

                {children}
            </div>
        );
    }
});
//模态框标题组件
ModalComponent.Title = React.createClass({
    render: function () {
        return (
            <h4
                id={this.props.id}
                className="modal-title"
            >
```

```
                {this.props.children}
            </h4>
        );
    }
});
//模态框体部组件
ModalComponent.Body = React.createClass({
    render: function () {
        return (
            <div className="modal-body">
                {this.props.children}
            </div>
        );
    }
});
//模态框底部组件
ModalComponent.Footer = React.createClass({
    render: function () {
        return (
            <div className="modal-footer">
                {this.props.children}
            </div>
        );
    }
});
//一个完整的示例模态框组件
var MyModal = React.createClass({
    getInitialState: function () {
        return {showModal: false};
    },
    onClose: function () {
        this.setState({showModal: false});
```

```
        },
        onOpen: function () {
            this.setState({showModal: true});
        },
        render: function () {
            return (
                <div>
                    <button  className="btn  btn-default"  onClick={this.
onOpen}>弹出模态框</button>
                    <ModalComponent          show={this.state.showModal}
ariaLabelledby={this.props.ariaLabelledby} onHide={this.onClose}>
                        <ModalComponent.Header  closeButton  aria-label=
"Close"  onHide={this.onClose}>
                            <ModalComponent.Title         id={this.props.
ariaLabelledby}>Modal 标题</ModalComponent.Title>
                        </ModalComponent.Header>
                        <ModalComponent.Body>
                            Modal 内容
                        </ModalComponent.Body>
                        <ModalComponent.Footer>
                            <button   type="button"   className="btn   btn-
default" onClick={this.onClose}>关闭</button>
                        </ModalComponent.Footer>
                    </ModalComponent>
                </div>
            );
        }
    });

    ReactDOM.render(
        <MyModal ariaLabelledby="modal-label"/>,
        document.getElementById("modal-component")
    );
```

7.6 综合实例

7.6.1 综合实例一

实例包中的 grid-ztree.html 文件给出了一个综合性的 React 组件实例。这个实例实现了按树形结构对 grid 内容进行筛选的功能，单击左边树的节点，右边的 grid 组件显示对应的过滤内容，是对前面所介绍组件的一个综合应用，其技术原理并不复杂，这里不再展开分析，读者可以自行查看分析源代码。下面附上该实例的运行效果图：

7.6.2 综合实例二

第二个综合实例是一个消息管理的实例，对应实例包文件夹为 chapter7/example-comprehensive，运行效果如下图：

整体界面分两个部分，上部用来展示现有消息列表，下部用来输入新的消息。单击上部的消息条目会使消息变灰并增加“(已读)”字样，单击条目右边的“×”按钮可以删除该条消息；在下部的文本输入框中输入消息后单击“发送”按钮，新增的消息将出现在消息列表中。

项目文件夹下的 js 文件实际上是经过 Babel 转译和打包之后的代码，不具有可读性。这个示例还用于第 13 章的单元测试，实际源代码位于实例包 chapter13/example-testing 文件夹，可以到该文件夹下查看源码。为便于测试代码对照，下面给出部分代码。

App 组件关键代码：

```
export default class App extends React.Component {
  render() {
    return (
      <div>
        <h2>消息列表</h2>
        <MessageList/>
        <AddMessage/>
      </div>
    );
  }
}
```

MessageList 组件部分代码：

```
export default class MessageList extends React.Component {
  constructor(props) {
```

```
    super(props);
    this.onChange = this.onChange.bind(this);
    this.state = MessageStore.getAll();
  }

  render() {
    const messageItems = this.state.messages.map(message => {
      return (
        <MessageItem key={message.id} message={message}/>
      );
    });
    return (
      <ul>{messageItems}</ul>
    );
  }
}
```

MessageItem 组件代码：

```
export default class MessageItem extends React.Component {
  constructor(props) {
    super(props);
    this.toggleRead = this.toggleRead.bind(this);
    this.deleteMessage = this.deleteMessage.bind(this);
  }

  toggleRead(e) {
    e.preventDefault();
    const message = this.props.message;
    if (!message.hasRead) {
      MessageStore.toggleRead(this.props.message.id);
      MessageStore.emitChange();
    }
```

```
    }

    deleteMessage(e) {
      e.preventDefault();
      MessageStore.deleteMessage(this.props.message.id);
      MessageStore.emitChange();
    }

    render() {
      const message = this.props.message;
      const messageReadClass = message.hasRead ? 'message-read' : '';
      let readText = message.hasRead && '(已读)';
      return (
        <li>
          <span className={`message-content ${messageReadClass}`}
   onClick={this.toggleRead}>{message.content}{readText}</span>
          <button className="delete" onClick={this.deleteMessage}> x
</button>
        </li>
      );
    }
  }
```

这个工程的特点是在界面组件之外，还维持了一个名为 MessageStore 的全局对象存放消息列表数据，MessageStore 封装对消息条目的操作，当数据发生变更时，还可以通过预先注册的回调函数通知目标对象。作为一个完整的项目，管理数据也是其必要的组成部分。React 下也有专门的框架提供范式的数据管理功能。

8

React插件

为了辅助 React 应用的开发，React 将经常用到的一些辅助工具性质的模块统一放置到 React.addons 集合中。React 官方声明，React.addons 系列模块所含功能是实验性的，将来可能经常发生变化。相比而言，内核变动比较少，要稳定一些。考虑本书内容的完整性，对此简要列表如下。

- TransitionGroup 和 CSSTransitionGroup 用于处理动画和过渡，如果用户自己实现这些操作将很简单，例如在一个组件移除之前执行一段动画。
- LinkedStateMixin 用于简化表单输入数据和组件 state 之间的双向数据绑定。
- cloneWithProps 用于实现 React 组件浅复制，同时改变组件的 props。
- update 是一个辅助方法，使在 JavaScript 中处理不可变数据变得更加容易。
- PureRenderMixin 用于在某些场景下的性能检测器。

以下插件只存在于 React 开发版（未压缩）：

- TestUtils 专门用于提供单元测试支持的辅助工具，用于编写测试用例。由于测试在开发阶段进行，因此该模块仅存在于未压缩版中。
- Perf 用于性能测评，以及帮助用户检查出可优化的功能点。

要使用某个插件，只能通过 npm 来安装。如：

```
npm install react-addons-pure-render-mixin
```

React 早期的版本使用 react-with-addons.js 来包含这些插件，现在已不再支持。

9

React实用技巧

在实际前端开发中，我们往往还会遇到一些具有共性的问题，针对其中的很多问题业界也提供相应的解决方案或工具。本章围绕实际工程问题列出了一些常用的技巧和工具，便于读者引用。

9.1 绑定React未提供的事件

由于 React 的事件是面向虚拟 DOM 的，因此，有些真实的浏览器 DOM 事件是 React 未提供的。如果要在 React 中响应这样的事件，需要在 componentDidMount 中获取对应的真实 DOM 元素绑定事件，并在 componentWillUnmount 中取消事件绑定。这种技巧也常用于 React 与 jQuery 结合使用时。参见下面的示例：

```
var Comp = React.createClass({
  getInitialState: function() {
    return {windowWidth: window.innerWidth};
  },

  onWindowResize: function(e) {
    this.setState({windowWidth: window.innerWidth});
  },

  componentDidMount: function() {
    window.addEventListener('resize', this. onWindowResize);
  },

  componentWillUnmount: function() {
    window.removeEventListener('resize', this. onWindowResize);
  },

  render: function() {
    return <div>当前 window 宽度: {this.state.windowWidth}</div>;
  }
});

React.render(<Comp />, document.body);
```

9.2 通过AJAX加载初始数据

通过 AJAX 加载数据是一个很普遍的场景。在 React 组件中如何通过 AJAX 请求来加载数据呢？首先，AJAX 请求的源 URL 应该通过 props 传入；其次，最好在 componentDidMount 函数中加载数据。加载成功，将数据存储在 state 中后，通过调用 setState 来触发渲染更新界面。

注意：

AJAX 通常是一个异步请求，也就是说，即使 componentDidMount 函数调用完毕，数据也不会马上就获得，浏览器会在数据完全到达后才调用 AJAX 中所设定的回调函数，有时间差。因此当响应数据、更新 state 前，需要先通过 this.isMounted() 来检测组件的状态是否已经 mounted。

下面是利用 GitHub 网站提供的 API 接口获取某个用户近况信息的例子。

```
var UserGist = React.createClass({
  getInitialState: function() {
    return {
      username: '',
      lastGistUrl: ''
    };
  },

  componentDidMount: function() {
    $.get(this.props.source, function(result) {
      var lastGist = result[0];
      if (this.isMounted()) {
        this.setState({
          username: lastGist.owner.login,
          lastGistUrl: lastGist.html_url
        });
      }
    }.bind(this));
  },

  render: function() {
    return (
      <div>
        {this.state.username}'s last gist is
```

```
          <a href={this.state.lastGistUrl}>here</a>.
        </div>
      );
    }
  });

  React.render(
    <UserGist source="https://api.github.com/users/octocat/gists" />,
    mountNode
  );
```

使用 jQuery 库所提供的 ajax 请求$.ajax 函数数据也存在一些问题，如兼容性问题就很令人头疼。React 推荐使用 fetch 库，其在 API 接口层面和 jQuery 类似，读者可以自行搜索相关资料，熟悉$.ajax 可以很快上手。

9.3 使用ref属性

在 React 中，ref 属性是特定用途的属性，将 ref 属性绑定到渲染函数中输出的任何组件上，就可以获得对应的组件实例，通过这个实例可以获得对应的实际 DOM 元素。

9.3.1 ref字符串属性

React 通常在组件上使用一个字符串识别符来作为 ref 属性。在渲染函数中给要渲染的子组件增加 ref 属性，如：

```
<input ref="myInput" />
```

在其他地方，如事件处理代码中，通过 this.refs 访问真正的组件。

```
var input = this.refs.myInput;
var inputValue = input.value;
```

完整示例：

```
var MyComp = React.createClass({
  getInitialState: function() {
    return {userInput: ''};
  },
  handleChange: function(e) {
    this.setState({userInput: e.target.value});
  },
  clearAndFocusInput: function() {
    // 清空输入的数据
    this.setState({userInput: ''}, function() {
      // 此处获得真实 DOM 并获得焦点
      this.refs.theInput.getDOMNode().focus();
    });
  },
  render: function() {
    return (
      <div>
        <div onClick={this.clearAndFocusInput}>
          单击获得焦点并清空
        </div>
        <input
          ref="theInput"
          value={this.state.userInput}
          onChange={this.handleChange}
        />
      </div>
    );
  }
});
```

在这个例子中，MyComp 组件的渲染函数渲染一个<input />组件，组件的实例通过 this.refs.theInput 获取，这个过程由 React 自动填充。值得注意的是：ref 属性对于 React 组件获得的是组件实例，而对于 HTML 获得的则是对应的真实浏览器 DOM 元素。对于前者，我们可以直接调用组件在类定义中公开的任何成员函数。

对于复合组件来说，这个引用会指向一个组件类的实例，进而可以调用任何该类定义的方法。如果需要访问该组件类对应的实际 DOM 节点，可以用 ReactDOM.findDOMNode 来找到实际的节点。不过并不推荐这种做法，因为这样做绝大多数情况下都会打破封装性，并增加了对真实浏览器 DOM 的依赖，一般都能找到更清晰的模式，使只在 React 模型中编写代码就能达到同样的效果。

9.3.2 ref回调函数属性

React 组件的 ref 属性也可以是一个回调函数，并且这个回调函数会在组件被挂载后立刻被执行。回调函数的参数对应组件实例的引用，这个回调函数可以立即使用组件，或者保存这个引用便于以后使用。上面的例子使用 ref 回调函数改写之后如下：

```
var MyComp = React.createClass({
  getInitialState: function() {
    return {userInput: ''};
  },
  handleChange: function(e) {
    this.setState({userInput: e.target.value});
  },
  clearAndFocusInput: function() {
    // 清空输入的数据
    this.setState({userInput: ''}, function() {
      // 此处获得真实 DOM 并获得焦点
      this.theInput.focus();
    });
```

```
    },
    render: function() {
      return (
        <div>
          <div onClick={this.clearAndFocusInput}>
            单击获得焦点并清空
          </div>
          <input
            ref={(compInstance)=>this.theInput=compInstance}
            value={this.state.userInput}
            onChange={this.handleChange}
          />
        </div>
      );
    }
  });
```

一旦引用的组件被卸载，或者 ref 属性本身发生了变化，原有的 ref 会再次被调用，此时参数为 null，这样可以防止内存泄露。如果和上面的例子一样使用内联的函数表达式，那么 React 每次更新（即调用渲染函数）时，都会得到不同的函数对象，之后每次更新时 ref 回调函数都会被调用两次：前一次参数是 null，后一次是具体的组件实例。

注意：

- 不要在任何组件的渲染函数或者被渲染函数调用的过程中访问 refs。
- 在 Google Closure Compiler 的高级模式下，不能访问用字符串形式声明的动态属性，即如果定义 ref='myRefString'，则必须以 this.refs['myRefString']的形式来访问引用。
- 对于 stateless function 来说，引用不会被加载，因为无状态组件并没有对应的实例。可以使用标准的复合组件来包装一个无状态组件，以在其上附加引用。

9.4 使用classNames.js

9.4.1 classNames介绍

在实际应用中，经常会遇到根据某些状态增加或更改组件属性中类名的情况，例如 Bootstrap 中的动态行为往往是由不同的 class 来控制，这就经常需要对 class 进行修改。看看下面的这段代码：

```
    var classString = 'content';
    if (this.state.isBgRed) {
        classString = classString + ' bg-red';
    } else {
        classString = classString + ' bg-green';
    }
```

这里实现了根据 isBgRed 状态切换不同 class 项的效果，但这样的代码语义不太清晰，也不容易维护，一旦状态控制变量变多，这样的代码将会变成一团乱麻。为了更好地满足这种类似的 class 动态切换的需求，可以使用 classNames 工具。针对这个问题以前还出现过 classSets 工具，功能与 classNames 相似，现已废弃不用。

使用 classNames 工具首先要导入这个模块：

```
    import classNames from 'classnames'
```

然后就可正常使用 classNames()函数了。前面的代码用 classNames 工具重写如下：

```
    let isBgRed = this.state.isBgRed;
    let classes = classNames ({' bg-red ':isBgRed, ' bg-green ': !
isBgRed });
```

9.4.2 classNames用法

classNames()函数可以接受任意数量的参数,并将参数连接成一个 class 字符串,参数可以是一个字符串或者一个对象。如果参数是对象,则对象的属性名(属性值为 false、0、null 或 undefine 的属性除外)也会加入到结果 class 字符串中。看下面的示例:

```
    classNames('foo', 'bar'); // 结果为“foo bar”
    classNames('foo', { bar: true }); //结果为“foo bar”
    classNames({ 'foo-bar': true }); // 结果为“foo-bar”
    classNames({ 'foo-bar': false }); //结果为空
    classNames({ foo: true }, { bar: true }); //结果为“foo bar”
    classNames({ foo: true, bar: true }); // 结果为“foo bar”

    // 多个不同类型的参数示例
    classNames('foo', { bar: true, duck: false }, 'baz', { quux: tru
e }); // 结果为“foo bar baz quux”

    // 忽略参数的示例
    classNames(null, false, 'bar', undefined, 0, 1, { baz: null }, '
'); // 结果为“bar 1”
```

如果参数是数组,则数组中的元素也被当作参数进行处理:

```
    var arr = ['b', { c: true, d: false }];
    classNames('a', arr); // 结果为“a b c”
```

9.4.3 在ES 6 中使用动态的classNames

ES 6 为 JavaScript 新标准语法,如果在工程中用到了 ES 6 语法(需要使用 Babel 进行转译),那么也可以结合 ES 6 中的动态属性特性来使用 classNames。如下面的例子:

```
let buttonType = 'primary';
classNames({ [`btn-${buttonType}`]: true });
```

9.4.4 多类名去重

当多个参数进行合并时，可能会遇到类名有重复的情况，这就需要进行去重处理。classNames 模块中的 dedupe 模块包含这方面的功能。dedupe 是 classNames 的增强版本，但性能要差些，因此，如非确实有去重需求，没必要使用它。这里只对其进行简要介绍。

要使用 dedupe，首先需要引入模块：

```
var classNames = require('classnames/dedupe');
```

或者在 html 文件中加入：

```
<script src="js/dedupe.js" type="text/JavaScript"></script>
```

使用方法与 classnames 一样，只是额外增加了自动去重的功能：

```
classNames('foo', 'foo', 'bar'); // 结果为“foo bar”
classNames('foo', { foo: false, bar: true }); // 结果为“bar”
```

9.5 使用Immutable.js

9.5.1 Immutable.js介绍

JavaScript 中的对象一般是可变的（Mutable），作为参数传递时使用的是引用。当声明一个对象的引用变量时，该引用变量直接指向原对象，如：

```
var foo={a: 1};
var bar=foo;    //此时 bar 与 foo 为同一对象
```

```
bar.b = 2;      //此时 foo.b 也为 2
```

其实这样的语义是不清晰的，可能我们只是想复制一个与 foo 一样的对象，结果却改变了原对象。尽管这样的共享变量机制可以节约内存，但在复杂场景下却往往是造成各种怪异问题的根源。Pete Hunt 说，共享的可变状态是罪恶之源（Shared mutable state is the root of all evil）。一般使用对象复制来避开这个问题，但马上就会遇到是浅复制还是深复制的选择，把问题复杂化了。使用同样来自名门 facebook 公司的 Immutable 模块来管理这样的数据是更好的选择。

使用 Immutable 来管理的数据对象具有只能创建不能更改的特性。对 Immutable 数据对象的任何修改、添加、删除操作都会返回一个新的 Immutable 对象，同时原对象依然可用且不变，这称为持久化数据结构（Persistent Data Structure）。为了避免深度复制，Immutable 使用了被称为结构共享（Structural Sharing）的策略，如果对象树中只有一个节点发生了变化，则只修改受到影响的父节点对它的引用，其他节点则与原对象共享，从而避免了深度复制带来的内存开销。

基于这样的策略，Immutable 还具有很多优秀的特性，比如支持回撤、并发安全、与函数式编程天然一致等，可以说 Immutable 是对数据管理的革新，其将产生更广泛的影响。如 React 就提倡把 this.state 当作只可创建不能更改的 Immutable，Redux 也推荐搭配使用 Immutable 来管理 state 数据等。

对于 Immutable 的具体用法，官网上有详细的文档。限于篇幅，本书主要列举了 Immutable 一些常用的、实用的用法，我们需要重点关注的是 Immutable 提供的持久化 List、Map 等用法，部分代码引自官方网站（http://facebook.github.io/immutable-js/）。

9.5.2 Immutable基本用法

1．创建 Immutable 对象

```
import Immutable from 'immutable';
```

```
    // 支持数据嵌套的创建方式
    var imtArray = Immutable.fromJS([1, {2,3}]) //根据数组创建
    var imtObj = Immutable.fromJS({a: 1, b:[2,3]}) //根据 JavaScript 对象创建
    // 不支持数据嵌套的创建方式
    var imtList = Immutable.List([1,2]) //根据数组创建，支持数据
    var imtMap = Immutable.Map({a: 1}) //根据 JavaScript 对象创建
```

2．从 Immutable 对象中提取 JavaScript 对象

```
    var jsObject = immutableData.toJS(); // 提取 JavaScript 对象
    var jsArray = imtArray.toArray(); // 提取数组对象
```

9.5.3 Immutable对象比较

Immutable 对象的比较有两种方式，一种是引用比较，一种是值比较。

1．引用比较

引用比较用来识别两个对象是不是同一个对象。使用操作符===进行比较，由于比较的是内存地址，所以速度很快。

```
    let map1 = Immutable.Map({a:1, b:2, c:3});
    let map2 = Immutable.Map({a:1, b:2, c:3});
    map1 === map2;           // 结果为 false
```

因为只比较内存地址，即使两个不同的对象内容一样也被认为是不相等的。如果需要进行内容上的比较，可以使用值比较的方式。

2．值比较

值比较用于判断两个数据对象的值是否相等，使用 Immutable.is()函数进行比较：

```
    Immutable.is(immutableObjectA, immutableObjectB);
```

Immutable.is()函数实质上比较的是两个对象的 hashCode 或 valueOf（对于 JavaScript 对象）。由于 Immutable 使用持久化数据结构存储对象，只要两个对象的 hashCode 值相等，两个对象的值就是一样的，有效地避开了对对象值的深度遍历，非常高效。

9.5.4 Immutable List用法

（1）创建 Immutable List。

```
Immutable.List(); //生成空的 Immutable List 对象
Immutable.List([1,2]); //生成 Immutable 数组对象，不支持嵌套
Immutable.fromJS([1,{2,3}]); //生成 Immutable 数组对象
```

（2）查看 List 的大小。

```
immutableA.size
immutableA.count()
```

（3）判断是否是 List。

```
Immutable.List.isList(x);
```

（4）在 React 组件中判断 propTypes 是否是 List 的写法。

```
React.PropTypes.instanceOf(Immutable.List).isRequired
```

（5）获取 List 索引的元素。

```
immutableData.get(0);
immutableData.get(-1); //当索引值为负数时为反向索引
```

（6）访问嵌套数组中的数据。

```
var imtNestedData = Imutable.fromJS({a: {b: {c:3}}});
imtNestedData.getIn(['a', 'b', 'c']); //结果为 3
```

（7）更新 List，其实就是根据原来的 List 对象创建一个新的 List 对象。

```
immutableA = Immutable.fromJS([0, 0, [1, 2]]);
immutableB = immutableA.set(1, 1);
immutableC = immutableB.update(1, (x) -> x + 1);
immutableC = immutableB.updateIn([2, 1], (x) -> x + 1);
```

（8）针对 List 的排序方法有 sort 和 sortBy。

```
immutableData.sort(function(a, b){
  if a < b then return -1;
  if a > b then return 1;
  return 0;
});
immutableData.sortBy((x) -> x);
```

（9）遍历。

```
immutableData.forEach(function(a, b){
  // 此处进行遍历操作
  return true; //如果返回值为 false 则终止遍历
});
```

（10）检索 List 中的元素。

```
immutableData.find((x) -> x > 1);  // 返回第一个匹配的元素
immutableData.filter((x) -> x > 1);  // 返回匹配的元素的数组
immutableData.filterNot((x) -> x <= 1); // 返回不匹配的所有元素的数组
```

9.5.5 Immutable Map用法

（1）创建 Immutable Map。

```
Immutable.Map();       // 创建空的 Immutable Map
Immutable.Map({a: 1});   // 根据对象创建 Immutable Map
Immutable.fromJS({a: 1}); // 根据对象创建 Immutable Map
```

（2）判断 Map，写法和 List 类似。

```
Immutable.Map.isMap(obj);
```

（3）获取 Map 中的数据。

```
immutableData.get('a');
```

通过 getIn 访问嵌套的 Map 中属性 a 值对象中 b 属性的值。

```
immutableData.getIn(['a', 'b']);
```

（4）更新对象。

```
immutableB = immutableA.set('a', 1);
immutableB = immutableA.setIn(['a', 'b'], 1);
immutableB = immutableA.update('a', (x) -> x + 1);
immutableB = immutableA.updateIn(['a', 'b'], (x) -> x + 1);
```

合并对象。

```
immutableB = immutableA.merge(immutableC);
```

（5）Map 的检索，与 List 相似。

```
data = Immutable.fromJS({a: 1, b: 2});
data.filter((value, key) -> value is 1);
```

判断属性是否存在要先转换为 JavaScript 的原生对象，再判断:

```
immutableData = Immutable.fromJS({key: null});
immutableData.has('key');
```

（6）分别获取 key 的数组和 value 的数组。

```
immutableData.keySeq();
immutableData.valueSeq();
```

9.6 与jQuery集成

毋庸置疑，jQuery 仍然是当前主流的 Java 工具库。尽管 React 很优秀，但它毕竟是新生事物，和庞大的 jQuery 生态资源相比，React 就显得很不足了。即使使用 React，我们也会因为各种各样的原因，再将它和 jQuery 结合到一起使用。如何延续 jQuery 的资源又结合 React 的技术呢？围绕这个问题,我们首先来总结一下 React 与 jQuery 的区别。

9.6.1 React与jQuery的区别

（1）DOM 操作方式不同：jQuery 主要操作的是实际 DOM 元素，React 操作的是虚拟 DOM。React 是数据驱动的，我们也很少需要操作 DOM，只需要关注数据的变化即可。另外，jQuery 中有专门针对不同浏览器的处理方案，React 则不再需要了，React 中包含了由虚拟 DOM 到真实 DOM 的转换模块，因此 React 不再需要考虑兼容性问题。

（2）元素选取方式有所不同：jQuery 通过 ID、class 等选择器选择元素。而在 React 中，因为有虚拟 DOM，所以 jQuery 的选取方式不再有效，React 通常使用 ref 方式来获取元素。

（3）渲染方式不同：jQuery 将更新 DOM 的职责交给用户，用户可以整体更新也可以局部更新，取决于具体实现。而 React 考虑的是整体更新机制，在实际更新的时候更新的是局部。

（4）事件处理方式不同：jQuery 实现了自己的一套事件处理逻辑，React 也有自己的一套事件处理系统。但总体来说，两者都是在原生 JavaScript 的基础上进行封装，而且都类似于原接口。

基于以上考虑，我们很难不做改动地将 jQuery 组件变为 React 组件。因此，我们重点考虑如何在 jQuery 中使用 React，以及如何在 React 中使用 jQuery。

9.6.2 在React中使用jQuery

要在 React 中使用 jQuery 的功能，首先要获得组件所对应的真实浏览器 DOM 元素，这点通过 ref 属性可以实现；其次要注意 jQuery 事件系统与 React 的不同，重点需要关注 componentDidMount 和 componentWillUnmount 函数，并在这两个函数内适配 jQuery 的生命周期。以下代码片段引自 React 官方示例。

```
    var BootstrapModal = React.createClass({
      componentDidMount: function() {
        var rootElem = this.refs.root; //获得真实的浏览器 DOM 节点
        // 调用 jQuery 方法，将节点内容转换为模态对话框
        $(rootElem).modal({backdrop: 'static', keyboard: false, show:
false});
        // 绑定 jQuery 事件，注意到 root 组件并没有在 React 中绑定事件
        $( rootElem).on('hidden.bs.modal', this.handleHidden);
      },
      componentWillUnmount: function() {
        $(this.refs.root).off('hidden.bs.modal', this.handleHidden);
      },
      close: function() {
        $(this.refs.root).modal('hide');
      },
      open: function() {
        $(this.refs.root).modal('show');
      },
      render: function() {
        var confirmButton = null;
        var cancelButton = null;
```

```
      if (this.props.confirm) {
        confirmButton = (
          <BootstrapButton
            onClick={this.handleConfirm}
            className="btn-primary">
            {this.props.confirm}
          </BootstrapButton>
        );
      }
      if (this.props.cancel) {
        cancelButton = (
          <BootstrapButton  onClick={this.handleCancel}  className=
"btn-default">
            {this.props.cancel}
          </BootstrapButton>
        );
      }
      return (
        <div className="modal fade" ref="root">
          <div className="modal-dialog">
            <div className="modal-content">
              <div className="modal-header">
                <button
                  type="button"
                  className="close"
                  onClick={this.handleCancel}>
                  &times;
                </button>
                <h3>{this.props.title}</h3>
              </div>
              <div className="modal-body">
```

```
              {this.props.children}
            </div>
            <div className="modal-footer">
              {cancelButton}
              {confirmButton}
            </div>
          </div>
        </div>
      </div>
    );
  },
  handleCancel: function() {
    if (this.props.onCancel) {
      this.props.onCancel();
    }
  },
  handleConfirm: function() {
    if (this.props.onConfirm) {
      this.props.onConfirm();
    }
  },
  handleHidden: function() {
    if (this.props.onHidden) {
      this.props.onHidden();
    }
  }
});
```

9.6.3 在jQuery中使用React

在 jQuery 中使用 React 与在 HTML 中使用 React 没有什么不同，我们只要声明一个

HTML 标签（通常是 div）作为 React 组件的容器，在初始化阶段调用 ReactDOM.render() 函数将组件挂接到该标签下即可。如下面的例子：

```
ReactDOM.render(
  React.createElement(HelloComponent, null),
  document.getElementById('reactContainer')
);
```

第二篇

React开发相关工具链

在实际工程开发中，仅掌握 React 组件基本知识是不够的。要提高开发质量和效率，还需要有一系列的工具支持，实现自动模块依赖管理、基于 JavaScript 新规范 ES 6 的使用、自动打包、开发中的代码热升级等这些功能，最好将这些工具进行整合，搭建一个完整的开发环境。

本篇围绕实际工程开发环境搭建，介绍了相关的开发工具链。具体包括 JavaScript 运行环境 Node.js、包管理工具 npm、ES 6 规范、规范检查工具 ESLint、转译工具 Babel 以及打包工具 webpack 的介绍和使用方法，重点介绍与 React 前端开发相关的、常用的知识点。熟练地掌握这些工具，尤其是 npm 和 webpack，是进行前端开发的基础，也是提升开发效率和质量的关键因素。

10

JS前端开发工具链

10.1 Node.js

Node.js（也称 node、nodejs）是一个 JavaScript 运行环境。Node.js 诞生于 2009 年，其诞生之初就没有历史包袱，几年来发展迅速，至今仍在不断更新完善。通常认为，JavaScript 的定位是作为浏览器脚本语言，Node.js 则颠覆了这个定位，将 JavaScript 变为一个可在服务器上运行的全栈语言。基于 Chrome V8 解释引擎，使用 Node.js 可方便地搭建响应速度快、易于扩展的网络应用。Node.js 的核心优势是使用事件驱动和非阻塞 I/O 模型，使其轻量且高效，非常适合在分布式设备上运行 I/O 密集型的实时应用。

Node.js 是用于后端服务器开发的，而 React 是前端框架，那么两者有何关系呢？

在 Node.js 未出现以前，JavaScript 只能运行在浏览器中，在前端开发中调试时，测试手段单一，一般也只能做一些简单的工作，要实现模块化更是困难。而 Node.js 的出现将 JavaScript 解放出来，也将以 JavaScript 为基础的前端解放出来，就有了包管理器、构建工具、打包工具的出现。这些工具不是运行在浏览器中，而是运行在 Node.js 环境中，只有最终生成的代码在浏览器中运行。

10.1.1 Node.js安装

这里简单介绍在 Windows 和 Linux 两种操作系统下安装 Node.js 的方法。

1．在 Linux 环境下的安装

Node.js 在 Linux 环境下的安装和使用都非常方便。下面以 Ubuntu 操作系统为例，介绍源码安装和仓库安装两种方式。

1）源码安装

（1）安装依赖包。

```
$sudo apt-get install g++ curl libssl-dev apache2-utils
$sudo apt-get install git-core
```

（2）在命令行下运行以下命令。

```
$git clone https://github.com/nodejs/node.git
$cd node
$./configure
$make
$sudo make install
```

（3）安装成功后，在命令行下输入以下命令查看版本，正常运行则安装成功。

```
$node -v
```

如果不用 Git 下载也可以直接从官网下载源码，地址为 https://nodejs.org/en/

download/。源码安装可以方便安装指定版本的 Node.js，只需要从 Git 克隆对应版本或从官网下载对应版本的源码包。

2）仓库安装

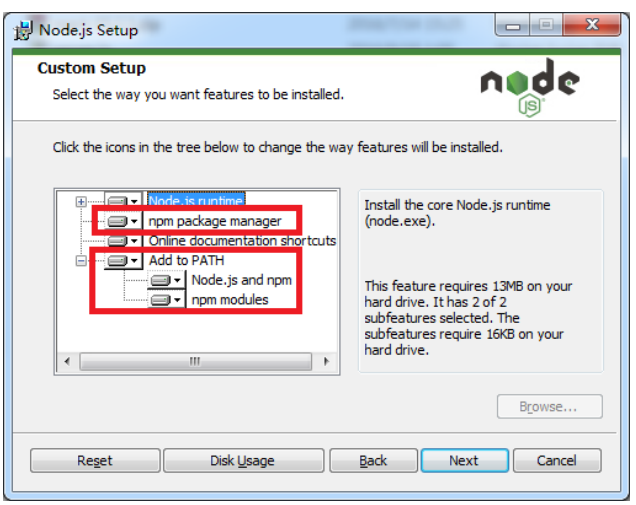
仓库安装是最简单的安装方法，但是进行仓库安装时，可能最新版本还没被收录，可以选择源码安装方式安装最新版本。

```
$sudo apt-get install nodejs
$sudo apt-get install npm
```

2．在 Windows 环境下安装

在 Windows 环境下可直接下载 Node.js 安装程序进行安装，下载地址为 https://nodejs.org/en/download/，安装程序分 32 位和 64 位两种类型，根据自己的操作系统选择对应的下载方式，按交互提示进行安装。

安装过程中，在 Custom Setup 界面要将 NPM 包管理器一起进行安装，并写入环境变量 PATH 中，如下图所示。



用户也可自行在环境变量 PATH 中添加 Node.js 的安装路径，以确保在命令行中可以调用 Node.js 命令。

在 Windows 环境下，也可使用 Windows 下的 UNIX 环境模拟器 Cygwin 安装 Node.js，但现在已很少采用这种方法。

10.1.2 Node.js使用

首先以一个最简单的 Node.js 程序（server.js）为例，内容如下：

```
const http = require('http');
const hostname = '127.0.0.1';
const port = 3000;
const server = http.createServer((req, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end('Hello World\n');
});
server.listen(port, hostname, () => {
  console.log('Server running at http://${hostname}:${port}/');
});
```

代码解释：

（1）全局方法 require()是用来导入模块的，一般直接把 require()方法的返回值赋给一个变量，在 JavaScript 代码中直接使用此变量即可使用该模块 。require(http)就是加载系统预置的 http 模块。

（2）http.createServer 是 http 模块的方法，创建并返回一个新的 Web Server 对象，并且给服务绑定一个回调，用以处理请求。

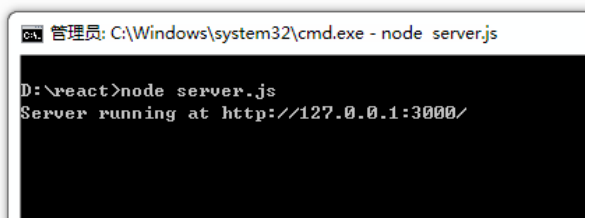
（3）通过 http.listen()方法可以让该 HTTP 服务器在特定端口监听。

（4）console.log()方法表示向控制台输出 log。

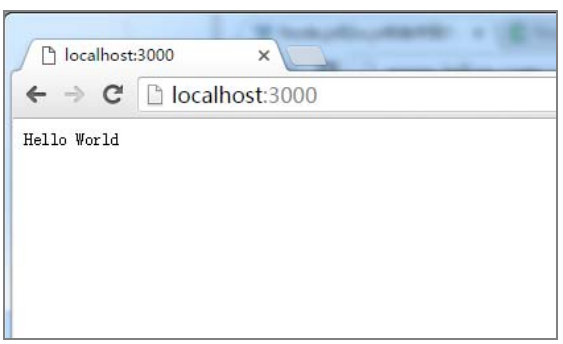
在命令行中运行：

```
$node server.js
```

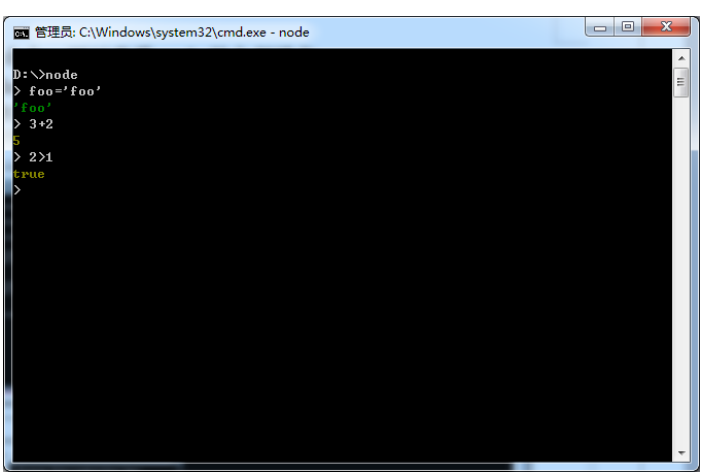
成功启动后可以看见 console.log()中的输出文本信息：



用浏览器打开 http://127.0.0.1:3000，效果如下：



直接在命令行中输入 node，即进入交互式控制台，在这里可以直接输入 JavaScript 代码，这和 Python 的控制台类似，在 Node.js 中称为 REPL(Read-Eval-Print-Loop)。



Node.js 还支持调试模式，例如调试模式运行 server.js：

```
$node debug server.js
```

进入调试模式启动程序后，程序在入口处停止，需要开发人员手动发起调试命令。主要调试命令包括 run、next、step、out 等，用户可在调试界面输入 help 查看全部命令。和在 IDE 中调试相比，利用命令行进行调试是非常不方便的，所以诞生了一些在 IDE（如 Eclipse）和浏览器（Chrome）中的调试工具，以方便程序员对代码进行调试。

在 React 前端开发中，并不需要直接使用 Node.js 编写服务器端代码，Node.js 主要用来搭建开发环境，例如构建工具、打包工具、代码检查、测试框架等，都是基于 Node.js 环境运行的，有时为更好地使用这些开发工具和框架，需要开发人员编写部分 Node.js 代码。

10.2 Node.js模块和包

10.2.1 模块

在 NPM 中一个模块就是一个 js 脚本文件，在这个脚本文件中可以声明导出一些由其他模块导入的变量、函数、对象等。

一个最简单的模块如下：

```
function hello(name) {
    console.log('Hello, '+ name);
}
exports.hello = hello;
```

以上这段代码定义了一个名为 hello 的模块，使用 hello 模块的代码如下：

```
var h = require('./hello');
h.hello('react');
```

这里需要将 hello.js 与 main.js 放到同一个文件夹下，运行结果如下：

```
$node main.js
Hello, react
```

10.2.2 包

一个包通常就是一个文件夹，但在这个文件夹下按包的格式约定包含了一些特殊的文件和文件夹。具体来说，一个包至少要包含包的描述文件 package.json 和具体的模块文件。

package.json 以 json 格式保存了包的描述、配置及依赖信息，了解 package.json 的内容是很重要的，完整的 package.json 包含三部分内容，一是包的基本属性，包括 name、version、description、keywords、homepage、bugs、license 等；二是与用户相关的属性，包括 author、contributors 等；三是包中文件及模块的描述，包括 files、main、bin、man、directories、directories.lib、directories.bin、directories.man、directories.doc、directories.example、repository、scripts、config、dependencies、URLs as Dependencies、Git URLs as Dependencies、GitHub URLs、Local Paths、devDependencies、peerDependencies、bundledDependencies、optionalDependencies、engines、engineStrict、os、cpu、preferGlobal、private、publishConfig、DEFAULT VALUES 等众多属性。

下面将对基础配置项进行简要说明。

- name：必选项，表示模块名称。命名时不能包含 js、node 和 url 中需要转义的字符，不能以.或_开头。
- version：必选项，表示模块的版本号。版本号格式为“主版本号（Major）.副版本号（Minor）.补丁版本号（Patch）”。

- main：必选项，模块入口文件相对路径（相对于模块根目录）。
- description：可选项，表示模块功能描述。
- keywords：可选项，数组类型，表示模块的关键字。
- author：可选项，表示发起者信息。示例如下：

```
"author":{
   "name": "fb",
   "email": "opensource+npm@fb.com"
}
```

- engines：可选项，所依赖的 Node.js 版本。示例如下：

```
"engines":{
   "node": ">= 0.8.0"
}
```

- repository：可选项，源码托管地址。示例如下：

```
"repository": {
   "type": "git",
   "url": "git+https://github.com/facebook/react.git"
}
```

- scripts：可选项，自定义包含在文件中的可执行命令，使用时输入 npm run <script>即可启动对应的命令。这个选项是常用的重要选项，在后面还会作详细介绍。
- dependencies 和 devDependencies：可选项，用于配置模块的生产环境依赖包和开发环境依赖包。后面会具体介绍。

10.3 npm模块管理器

npm 是 Node.js 的包管理工具，主要针对代码包和依赖的下载、使用、更新、卸载、上传和部署提供了整套解决方案，是当今 JavaScript 开发必不可少的工具。npm 的官网地址是 npmjs.org。随着前端和 Node.js 的发展和流行，目前，npm 社区软件包的数量已超越 C、C++，已然成为全球最大的代码工厂。

npm 常见的使用场景有以下几种：

- 从 npm 服务器下载第三方包到本地使用，如 react-dom。
- 从 npm 服务器下载第三方包并安装为可以执行的命令行程序供本地使用，如 babel。
- 将自己编写的包或命令行程序上传到 npm 服务器供他人下载使用。

10.3.1 npm安装

在安装 Node.js 的时候，默认安装 npm 包管理器，不需要再另外进行安装。在命令行中输入“npm –v”可以测试是否安装成功，出现版本提示表示成功安装：

```
$ npm -v
2.15.5
```

npm 既是模块管理的命令行工具，也是 Node.js 下的一个模块，实际上 npm 命令只不过是 node npm 命令的快捷方式而已。

10.3.2 npm初始化

前面已经介绍过 Node.js 包的描述文件是 package.json，该文件描述了一个 Node.js 包的所有相关信息，包括作者、简介、包依赖、构建等。

package.json 的格式必须是严格的 json 格式，通常并不需要自己创建该文件，编写模块的时候，首先使用 npm init 命令，通过交互式的命令，就可以自动生成一个 package.json 文件，里面包含了一些常用的字段信息。使用 npm init 交互式命令生成 package.json 的过程如下：

```
    $npm init
    This utility will walk you through creating a package.json file.
    It only covers the most common items, and tries to guess sensible
defaults.

    See 'npm help json' for definitive documentation on these fields
    and exactly what they do.

    Use 'npm install <pkg> --save' afterwards to install a package and
    save it as a dependency in the package.json file.

    Press ^C at any time to quit.
    name:  react（输入项目名称）
    version: (1.0.0)  （输入项目版本）
    description: react description（输入项目描述）
    entry point: (index.js) （输入项目入口文件名）
    test command: test（输入项目测试命令）
    git repository: （输入项目的 git 地址）
    keywords: react（输入项目关键字）
    author: react book author（输入项目作者）
    license: MIT（输入项目协议）
    About to write to D:\react\package.json: (显示生成 package.json 文件
的位置及内容)
```

```
{
  "name": "react",
  "version": "1.0.0",
  "description": "react description",
  "main": "index.js",
  "scripts": {
    "test": "test"
  },
  "keywords": [
    "react"
  ],
  "author": "react book author",
  "license": "MIT"
}

Is this ok? (yes)
```

运行成功后 npm 将在对应目录生成 package.json 文件。

10.3.3 npm安装模块

项目基本信息已经生成后，要安装项目所需要的依赖。访问网址 https://www.npmjs.com/，可以搜索 npm 仓库中提供了哪些模块。也可以在命令行中运行 npm search <module Name>来进行模块搜索。推荐使用 https://www.npmjs.com/网站进行搜索，该网站收录了相关模块介绍、使用说明等诸多信息。

现在以开发 react 所需依赖为例，在命令行中运行：

```
$npm install --save react react-dom babel-preset-react
```

安装好之后，react、react-dom 和 babel-preset-react 包就放在当前目录下的 node_modules 目录中，同时这些包的相关依赖包也会一并下载，这也体现了 npm 强

大的自动依赖管理机制。在代码中要使用该模块，只需要引用 React 即可，不用考虑其存放位置。

此处我们使用--save 参数，将依赖关系也写入到了 packages.json 文件中。运行结束后，查看 package.json 文件，出现对应的依赖内容及版本号：

```
"dependencies": {
   "babel-preset-react": "^6.16.0",
   "react": "^15.3.2",
   "react-dom": "^15.3.2"
 }
```

同时，npm 生成 node_modules 文件夹，存储依赖的具体内容。node_modules 文件夹是树形结构，结构与依赖关系一致。

在向其他开发者分享项目工程时，只需提供 package.json 文件，不需要提供 node_modules 文件夹。其他开发者使用或在其他地点运行本项目时，在命令行运行 npm install，npm 将自动根据 package.json 文件中的 dependencies 内容下载对应的依赖，并生成 node_modules 文件夹。

实际上 npm 模块安装分为全局模式和本地模式。以上介绍的是本地模式下的模块安装，模块包都会下载到当前目录下的 node_modules 文件夹下。一般情况下，npm 也推荐以本地模式运行，这样可以避免版本冲突。全局安装一般都是供命令行使用的，直接通过 require()的方式是没有办法调用全局安装的包的。在全局模式下，模块包默认被安装到 Node.js 安装目录下的 node_modules，这个默认的位置也是可以配置和修改的。

全局模式安装需要带-g 参数，如下：

```
npm install react –g
```

安装模块后，如何在代码中使用呢？在代码中引用模块的方式是使用 AMD 模块管理方式，例如：

```
var react = require('react');
```

目前，基于 JavaScript 语言的最新规范 ES 6 发布后，业界都倾向于使用 ES 6 中标准的模块引用语法，关于 ES 6 语法的内容在后面的章节会详细介绍。这里先看看其使用方式：

```
import {React} from 'react';
```

全局安装的模块不能直接引用，需要 link 到当前目录下才能正常引用。

10.3.4 使用cnpm

由于默认软件源在国外，所以在使用 npm 时经常遇到无法连接不能下载的情况。这时可以使用淘宝定制的 cnpm 命令行工具代替默认的 npm。

cnpm 本身也是一个 Node.js 模块，可以通过 npm 下载安装，安装命令是：

```
$ npm install -g cnpm --registry=https://registry.npm.taobao.org
```

或者直接添加参数 alias：

```
alias cnpm="npm --registry=https://registry.npm.taobao.org \
--cache=$HOME/.npm/.cache/cnpm \
--disturl=https://npm.taobao.org/dist \
--userconfig=$HOME/.cnpmrc"
# Or alias it in .bashrc or .zshrc
$ echo '\n#alias for cnpm\nalias cnpm="npm --registry=https://registry.npm.taobao.org \
  --cache=$HOME/.npm/.cache/cnpm \
  --disturl=https://npm.taobao.org/dist \
  --userconfig=$HOME/.cnpmrc"' >> ~/.zshrc && source ~/.zshrc
```

安装模块：cnpm 的使用方法与 npm 完全相同，最大的区别是 cnpm 是从 registry.npm.taobao.org 安装所有模块。使用 cnpm 进行模块安装时，如果发现 registry.npm.taobao.org 上还没有要安装的模块，就会自动在后台进行同步，并且会让我们从官方 npm(registry.npmjs.org)进行安装。下次再安装这个模块的时候，就会直接从淘宝 npm 安装了。

```
$ cnpm install <Module Name>
```

同步模块：直接通过 sync 命令同步一个模块，只有 cnpm 命令行才有此功能。

```
$ cnpm sync connect
```

当然，也可以直接通过 Web 方式来同步/sync/connect。

```
$ open https://npm.taobao.org/sync/connect
```

10.3.5 npm常用命令

下面来认识一下 npm 常用命令，并针对主要命令做具体介绍。

1．npm install <module Name >

模块安装完毕后会产生一个 node_modules 目录，其目录下就是安装的各个模块。在安装的过程中，模块所需的依赖模块也将一同被安装。

以安装 express 模块为例：

```
$npm install express
```

默认情况下，npm 会安装 express 的最新版本，也可以通过添加版本号的方式安装指定版本，如 npm install express@3.0.6。

Node.js 模块的安装分为全局模式和本地模式。一般情况下，以本地模式运行，模块会被安装到项目工程本地的 node_modules 下。在全局模式下，模块将被安装到 Node.js 安装目录的 node_modules 下。

全局安装命令为：

```
$npm install -g <module Name>
```

可以使用$npm set global=true 命令指定按照全局模式安装，$npm get global 命令可以查看当前使用的安装模式。需要注意的是，全局安装一般都是供命令行使用的，直接通过 require()方式是没有办法调用全局安装的模块的。如需要调用全局模

块，需要使用$npm link 命令创建链接，相当于创建一个从本工程模块目录到全局模块目录的软链接。命令如下：

```
$npm link <module Name>
```

此外，在模块安装的同时，可以将模块依赖内容写入到工程的 package.json 依赖属性中，命令如下：

```
$npm install <module Name > [-save|-save-dev|-save-optional]
```

添加-save 参数，安装的模块的名字及其版本将被添加到 package.json 的 dependencies 选项中。

添加-save-dev 参数，安装模块的名字及其版本将被添加到 package.json 的 devDependencies 选项中。如果只需要下载使用某些模块，而不下载这些模块的测试和文档框架，可以使用这个参数。

添加-save-optional 参数，安装模块的名字及其版本将被添加到 package.json 的 optionalDependencies 选项中。如果要求即使依赖项安装失败或没有找到，npm 仍会继续执行，可以使用该参数。

如果项目中已经存在 package.json 文件，直接使用 npm install 方法就可以根据该文件的依赖配置安装所有依赖模块，这样在代码提交和分享时，就不用提供 node_modules 文件夹了，只需要提供一个 package.json 描述文件。

2．npm view <module Name>

查看 node 模块的 package.json 文件。如果想要查看 package.json 文件下某个标签的内容，可以使用$npm view <module Name> <label Name>。

- $npm view <module Name> dependencies：查看包的依赖关系。
- $npm view <module Name> repository.url：查看包的源文件地址。
- $npm view <module Name> engines：查看包所依赖的 Node.js 的版本。

3．npm list

查看当前项目下已安装的包。

npm 模块的搜索是从代码执行的当前目录开始的，搜索结果取决于当前使用的目录中 node_modules 下的内容。

- $npm list parseable=true：可以目录的形式展现当前安装的所有包。

4．npm help

查看帮助命令。

- $npm help json：显示 package.json 的帮助文档。
- $npm help folders：显示 npm 目录结构的帮助文档。

5．npm rebuild <module Name>

用于更改包内容后进行重建。

6．npm outdated

检查包是否已经过期，此命令会列出所有已经过时的包，据此可以及时进行包的更新。

7．npm update <module Name>

更新 node 模块。

可以使用此命令更新 npm 自身：

```
$npm update -g npm
```

8．npm uninstall <module Name>

卸载 node 模块。

9．npm search <package Name>

安装一个 npm 包或者发布一个 npm 包时，都要检验某个包名是否已经存在，此时就要使用此命令。

10．npm init

引导创建一个 package.json 配置文件，包括名称、版本、作者等信息。

11．npm root

查看当前包的安装路径。

- $npm root -g：可以查看全局的包的安装路径。

12．npm -v

查看 npm 的安装版本。

13．npm publish

将模块发布到 www.npmjs.org 仓库上。

发布之前需要先在 www.npmjs.org 上注册，然后再执行 npm adduser 命令添加账号。

```
$npm adduser
Username: （输入用户名）
Password: （输入密码）
Email: （输入邮箱）
```

10.3.6 自定义脚本

npm 不仅可以用于模块管理，还可以用于执行脚本。开发者也可以在 package.json 文件中自定义脚本。

```
    "scripts": {
       "start" : "node server.js"
       "build": "webpack",
    }
```

在命令行运行：

```
    npm run build
```

该命令在执行中会自动调用 webpack 命令；同理，运行 npm run start 命令也会自动运行 node server.js 命令。为了方便，npm 内置了两个简写的命令：npm test 和 npm start。npm test 命令等同于执行命令 npm run test，npm start 命令等同于执行命令 npm run start。

开发者可以根据需要编写自定义脚本，例如下面这种复杂的脚本：

```
    "build-js": "browserify browser/main.js | uglifyjs -mc > static/bundle.js"
    "build": "npm run build-js && npm run build-css"
```

为了适应更复杂的场景，npm run 为每条命令提供了 pre-和 post-两个钩子。以 npm run test 为例，如果在 scripts 字段定义了 pretest 和 posttest：

```
    "scripts": {
       "test": "mocha test/",
       "pretest": "echo test start!",
       "posttest": "echo test end!"
    }
```

则会先执行 pretest 脚本，再执行 test 脚本，最后再执行 posttest 脚本。

10.4 ES 6 规范简介

ECMA（European Computer Manufacturers Association，欧洲计算机制造商协会）是制定信息传输与通信的国际化标准组织。ECMAScript 是 ECMA 制定的标准化脚本语言。ECMAScript 6（ES 6）是 JavaScript 语言的下一代标准，它的目标是使 JavaScript 语言可以用来编写复杂的大型应用程序，成为企业级开发语言。标准的制定者计划每年发布一次标准，以年份作为版本。因为 ES 6 的第一个版本是在 2015 年发布的，所以又称为 ECMAScript 2015（ES 2015）。

目前，虽然前端浏览器尚不直接支持 ES 6 语法，但是越来越多的开发者已经开始使用 ES 6，借助转换工具和打包工具，可以很方便地将其转换为浏览器识别的 ES 5 语法。所以有必要在这里简单介绍一下 ES 6 的新特征。

10.4.1 ES 6 语法简介

1. let 与 const 关键字

let 与 const 这两个关键字的用途与 var 类似，都是用来声明变量的，但在实际运用中都有各自的特殊用途。let 实际上是为 JavaScript 新增了块级作用域，用它所声明的变量，只在 let 命令所在的代码块内有效。const 也用来声明变量，但是声明的是常量。一旦声明，常量的值就不能改变。

先来看 let 的例子：

```
var name = "react";
while (true) {
    var name = "let";
    console.log(name);  //let
```

```
    break;
}
console.log(name);  //let
```

由于 ES 5 没有块作用域，使用 var 声明的变量 name 两次输出都是“let”，这带来很多不合理的场景，经常出现程序员难以发现的错误，也让很多习惯其他语言的程序员对 JavaScript 很是费解。而使用 let 声明，则可以避免这个问题，while 块内定义 name 变量的值是“let”，而块外的变量值仍为“react”：

```
let name = "react";
while (true) {
    let name = "let";
    console.log(name);  //let
    break;
}
console.log(name);  //react
```

const 关键字声明的是常量，声明后不可修改，如果要改变 const 声明的常量，浏览器就会报错。

```
const PI = Math.PI
```

const 有一个很好的应用场景，就是当我们引用第三方库时声明的变量，用 const 来声明可以避免因重命名而出现的 bug：

```
const react= require('react')
```

2. 类的支持

类的支持主要包括 class、extends、super 三个关键字，涉及了 ES 5 中最令人头疼的几个部分：原型、构造函数、继承。ES 6 对类的支持使其写法更接近其他面向对象编程语言的语法，更加清晰易理解。

```
class Animal { //类的定义
    constructor(name) {//ES6 中的构造器
```

```
        this.name = name;
    }
    sayMyName() {
        console.log('My name is '+this.name);
    }
}
class Programmer extends Animal { //类的继承
   constructor(name) {
super(name); //直接调用父类构造器进行初始化
    }
    program() {
        console.log("I'm coding...");
    }
}

let animal=new Animal('kitty'),
let programmer=new Programmer('react');
animal.sayMyName();   //“My name is kitty”
programmer.sayMyName();  //“My name is react”
programmer.program();   //“I'm coding...”
```

在上面的代码中，首先用 class 定义了一个类，类内部有一个 constructor()方法，即构造方法，而 this 关键字则代表实例对象。简单地说，constructor 内定义的方法和属性是实例对象自己的，而 constructor 外定义的方法和属性则是所有实例对象可以共享的。定义类时可以通过 extends 关键字实现继承，上面定义了一个 Programmer 类，该类通过 extends 关键字，继承了 Animal 类的所有属性和方法。

super 关键字，指代父类的实例（父类的 this 对象）。子类必须在 constructor()方法中调用 super()方法，否则新建实例时会报错。因为子类没有自己的 this 对象，而是继承父类的 this 对象，然后对其进行加工。如果不调用 super()方法，子类就得不到 this 对象。

3. 箭头操作符

ES 6 中新增的箭头操作符=>类似于 C#或 Java 中的 lambda 表达式，大大简化了函数的书写，操作符左边为输入的参数，右边是进行的操作及返回的值。箭头操作符是 ES 6 最常用的一个新特性。来看一个简单的对比：

```
function(i){ return i + 1; } //ES 5
(i) => i + 1 //ES 6
```

再看一个复杂一点的例子，在 JavaScript 中回调是经常的事，而回调一般又以匿名函数的形式出现，每次都需要写一个 function()，甚是烦琐。引入箭头操作符后便可以方便地写回调了：

```
var array = [1, 2, 3];
//ES5
array.forEach(function(v, i, a) {
   console.log(v);
});
//ES6
array.forEach(v = > console.log(v));
```

4. 字符串模板

字符串模板是非常有用的，当我们要插入大段的 html 内容到文档时，常常用“+”连接字符串和变量，写法烦琐又容易出错。先看下面一段代码：

```
$("#result").append(
  "There are <b>" + basket.count + "</b> " +
  "items in your basket, " +
  "<em>" + basket.onSale +
  "</em> are on sale!"
);
```

使用 ES 6 的新特性模板字符串``后，写法如下：

```
$("#result").append(`
  There are <b>${basket.count}</b> items
   in your basket, <em>${basket.onSale}</em>
  are on sale!
`);
```

字符串模板用反引号`做起止标示，用${}引用变量，而且所有的空格和缩进都会被保留在输出之中。

5. 解构

ES 6 允许按照一定模式，从数组和对象中提取值，对变量进行赋值，这被称为解构（destructuring）。

看下面的 ES 5 的例子：

```
let cat = 'kitty'
let dog = 'benben'
let zoo = {cat: cat, dog: dog}
console.log(zoo)  //Object {cat: "kitty", dog: "benben"}
```

用 ES 6 的写法：

```
let cat = 'kitty'
let dog = 'benben'
let zoo = {cat, dog}
console.log(zoo)  //Object {cat: "kitty", dog: "benben"}
```

反过来可以写为：

```
let dog = {type: 'animal', amount: 2}
let { type, amount} = dog
console.log(type, amount)  //animal 2
```

6. 参数默认值和不定参数

在 ES 中，以调用 animal()方法为例，当忘记为传入参数赋值时，通常使用以下方式为变量赋默认值：

```
function animal(type){
    type = type || 'cat'
    console.log(type)
}
```

type = type || 'cat'这句话在 type 未赋值时，将被赋值为“cat”。如果用 ES 6 可以直接写为：

```
function animal(type = 'cat'){
    console.log(type)
}
```

不定参数是在函数中使用命名参数时接收不定数量的未命名参数。在 ES 5 中可以通过 arguments 变量达到这一目的。在 ES 6 中的例子如下：

```
function animals(...types){
    console.log(types)
}
animals('cat', 'dog', 'fish')  //["cat", "dog", "fish"]
```

10.4.2 ES 6 模块管理

早期的 CommonJS 和 AMD 是用于 JavaScript 模块管理的两大规范，前者定义的是模块的同步加载，主要用于 Node.js；后者定义的是模块的异步加载，通过 RequireJS 等工具适用于前端。在 ES 6 标准中，JavaScript 提供了原生的模块(module)支持，可以非常方便地将不同功能的代码分别写在不同文件中，各模块只需导出公共接口部分，然后通过模块导入方式可以在其他地方使用。

一个 ES 6 模块就是一个 JavaScript 代码文件，一个模块看起来就和普通的脚本

文件一样，只是多了 import 和 export 等关键字。下面对 ES6 模块的使用做一个简单的介绍。

首先是导出（export）。在模块中声明的任何内容默认都是私有的，如果需要向其他模块开放，必须导出那部分代码。最简单的方法是添加 export 关键字，export 关键字可以添加在 function、class、var、let 或 const 前面。

以下面的 exportmodule.js 为例：

```
export function Efunction(options) {
  ...
}

export class EClass{
  ...
}
```

导出之后其他模块就可以使用这个模块了，只需要在其他文件中导入（import）这个模块并且使用相关的函数：

```
import {Efunction} from "exportmodule.js";

function run() {
    var func = Efunction('hi modules');
}
```

要同时导入多个模块中的接口，可以这样写：

```
import {Efunction, Efunction1} from "exportmodule.js";
```

运行一个包含 import 声明的模块时，被引入的模块会先被导入并加载，然后根据依赖关系，每一个模块的内容会使用深度优先的原则进行遍历。重复引用的模块将会被忽略。

10.4.3 基于ES 6 语法的React组件写法

使用 ES 6 来编写 React 组件已经是大势所趋，本书后面章节的很多实例均使用 ES 6 规范进行编写。为了让用户掌握使用 ES 6 编写 React 组件的方法，下面给出了一个实例。对应的实例包为 chapter10/example-es6。

```
const contextTypes = {
  book: React.PropTypes.string
}

const defaultProps = {
  book:"React 前端技术与工程实践"
}

class App extends React.Component {
  constructor(props, context) {
    super(props, context);
    this.handleClick = this.handleClick.bind(this);
  }

  handleClick(event) {
    event.preventDefault();
    alert("received event: " + event);
  }

  render() {
    return (
      <div onclick={this.props.handleClick}>
        我和《{this.props.book}》有个约会
      </div>
    );
  }
```

```
}
App.contextTypes = contextTypes;
App.defaultProps = defaultProps;

export default App;
```

ES 6 是采用类继承的方式来定义组件类而不是以函数的方式定义。另外，defaultProps 和 PropTypes 的声明类似于定义静态变量的方式，相比于传统的getDefaultProps()函数这样的写法，ES 6 在语义上更清晰。事件响应函数的 bind 方式写法看起来也有差别，但其实质是一样的，在 ES 6 中有构造函数可以进行初始绑定，而传统写法没有构造函数，所以在写法上有区别。

```
import React from 'react';

var App = React.createClass({
  propTypes: {
      book: React.PropTypes.string
  },

  getDefaultProps: function() {
    return {book: "React 前端技术与工程实践"};
  },

  handleClick: function(event) {
    event.preventDefault();
    alert("received event: " + event);
  },

  render: function() {
    return (
      <div onClick={this.handleClick.bind(this)}>
        我和《{this.props.book}》有个约会
```

```
        </div>
      )
    }
});

export default App;
```

10.5 ESLint工具

10.5.1 ESLint介绍

ESLint 是一个 JavaScript 代码静态检查工具，可以检查 JavaScript 的语法错误，提示潜在的 bug，并可以有效提高代码质量。ESLint 不但提供一些默认的规则，也提供用户自定义规则来约束所写的 JavaScript 代码。

ESLint 提供以下支持：

- ES 6。
- AngularJS。
- JSX。
- Style 检查。
- 自定义错误和提示。

ESLint 提供以下校验：

- 语法错误校验。
- 不重要或丢失的标点符号，如分号。

- 运行不能到达的代码块。
- 未被使用的参数提醒。
- 漏掉的结束符，如}。
- 确保样式的统一，如 sass 或 less。
- 检查变量的命名。

10.5.2 安装和使用

使用 npm 进行全局安装：

```
$ npm install -g eslint
```

在工程根目录输入：

```
$ eslint --init
```

该命令可以帮助用户生成配置文件.eslintrc.js。运行后在命令行中通过交互式命令生成配置文件。这里推荐使用交互式问答方式进行，ESLint 会询问是否使用 ES 6、JSX、React、CommonJS 等方式，用户只需回答问题即可，回答完成后在本目录将产生.eslintrc.js 文件。

需回答的问题包括：

Are you using ECMAScript 6 features? 你是否使用 ES 6 特性？

Are you using ES 6 modules? 你是否使用 ES 6 模块？

Where will your code run? 代码在哪里运行？

Do you use CommonJS? 你是否使用 CommonJS？

Do you use JSX? 你是否使用 JSX？

Do you use React? 你是否使用 React？

What style of indentation do you use? 你使用什么缩进？

What quotes do you use for strings? 你使用单引号还是双引号表示字符串？

What line endings do you use? 你使用哪种换行方式？

Do you require semicolons? 你是否使用分号？

What format do you want your config file to be in? 配置文件保存为什么形式？

初始化配置后，添加一个最简单的待检查的文件 add.js：

```
function add (a, b) {
    return a + b;
}
```

执行检查命令：

```
$ eslint add.js
```

命令执行后，检查结果将会出现在命令行界面中。

可以手动在参数中指定配置文件：

```
$ eslint -c config.json add.js
```

10.5.3 配置

使用 ESLint 时要注意规则配置，以引号的规则为例：

```
"quotes": [2, "double"]
```

第一部分是规则名：此处表示引号规则。

第二部分表示级别：0——不验证；1——警告；2——错误。此处配置表示引号必须使用双引号，否则将报告错误。

以下三种方法可以使用 ESLint 的配置规则：

- 使用.eslintrc 文件（支持 JSON 和 YAML 两种语法）。
- 在 package.json 中添加 eslintConfig 配置块。
- 直接在代码文件中定义。

配置文件中：

```
"extends": "eslint:recommended"
```

表示使用默认的规则进行检查，如果启用自定义规则需要将此设置去掉。

Eslint 支持在文件中使用注释进行标记来关闭指定的规则检查。

关闭 Eslint 检查的方法如下：

```
/*eslint-disable */
alert('foo');
/*eslint-enable */
```

指定对某些规则不进行检查的方法如下：

```
/*eslint-disable no-alert, no-console */
alert('foo');
console.log('bar');
/*eslint-enable no-alert */
```

ESLint 的详细规则说明可参考官方文档 http://eslint.org/docs/rules/，这里不再详述。

10.5.4 React检查

ESLint 支持对 JSX 和 React 的相关语法检查。目前，ESLint 已经原生支持对 JSX 语法的检查，对 React 的检查则是通过 eslint-plugin-react 插件实现的。

```
$npm install eslint-plugin-react
```

配置文件如下：

```
    "parserOptions": {
        "ecmaFeatures": {
            "experimentalObjectRestSpread": true,
            "jsx": true
        },
        "sourceType": "module"
    },
    "plugins": [
        "react"
    ]
```

一般不需要手动安装并编写配置文件，通过 eslint --init 完全可以自动完成这些操作。只需要在回答 Do you use JSX? 和 Do you use React? 两个问题时回答“Yes”，配置文件便可以自动生成，并自动安装 eslint-plugin-react。

10.6 Babel工具

当前各主流浏览器均已经支持 ES 5 规范。ES 6 规范虽然已经发布，但是直接支持 ES 6 的解释器暂时还不多。如果要用 ES 6 写代码需要使用转译工具，将 ES 6 的代码转译成 ES 5 的代码。在转译工具中，Babel 无疑是当前最强大、最广泛的一个工具，它除了转译 ES 代码之外，还提供了很多其他功能，如 JSX 转译等，甚至还支持新发布的 ES 7 规范。所以即使将来 ES 6 不再需要转译就能直接运行，Babel 也是值得学习的。

Babel 作为转译工具，可以以多种形态发挥转译作用，如在命令行中使用、实时转译、在浏览器中嵌入等。但无论哪种形态，都必须要配置.babelrc 文件。

10.6.1 配置.babelrc文件

Babel 配置文件.babelrc 放在项目的根目录下。要正确使用 Babel，首先要配置好.babelrc 文件。.babelrc 文件也是以 json 格式声明转译规则和插件配置，基本内容如下：

```
{
  "presets": [],
  "plugins": []
}
```

其中 presets 字段指定转译规则，官方提供的规则集见下表。注意，Babel 将 ES 6 也称为 ES 2015，另外，Babel 也支持 ES 7，由于 ES 7 是一个提案，不同的阶段包含了对 ES 7 不同程度的支持，这里不详细展开。一般我们使用 stage-2。

规 则 名	引 用 名	说　明
babel-preset-es2015	es2015	ES 2015 转译规则
babel-preset-react	react	react 转译规则，含 JSX 支持
babel-preset-stage-0	stage-0	ES 7 阶段 0 转译规则
babel-preset-stage-1	stage-1	ES 7 阶段 1 转译规则
babel-preset-stage-2	stage-2	ES 7 阶段 2 转译规则
babel-preset-stage-3	stage-3	ES 7 阶段 3 转译规则

每种规则都需要先安装才能正确使用，由于这些转译规则一般都是用于开发时，所以安装时应使用--save-dev 选项。如 babel-preset-es2015 安装命令如下：

```
npm install --save-dev babel-preset-es2015
```

在.babelrc 文件中可以加入多个规则，如：

```
{
  "presets": [ "es2015", "stage-2", "react" ],
  "plugins": []
}
```

plugins 字段主要用于配置各种插件，使用插件前需先安装好插件所对应的模块。

10.6.2 命令行转译工具：babel-cli

Babel 提供 babel-cli 模块，用于通过命令行完成转码。如果把它作为一个独立的命令行工具使用，应将其作为全局模块安装，即安装时应加上-g 选项。安装命令如下：

```
npm install -g babel-cli
```

Babel 命令行基本用法如下：

```
babel [options] <files ...>
```

这里 files 指定要转译的文件；option 为各种选项，常用的有-h、-f、-o 等选项。下面是 Babel 的常用场景。

- 将转译结果直接打印到标准输出，通常是屏幕。

```
babel example.js
```

- 将转译结果写入到文件，使用-o 或--out-file 选项指定输出文件。

```
babel -o result.js example.js
```

或：

```
babel --out-file result.js example.js
```

- 对目录下的所有文件转译，使用-d 或--out-dir 选项指定输出目录。

```
$ babel src --out-dir lib
```

或：

```
$ babel src -d lib
```

- 生成 source map 文件，使用-s 选项。

```
$ babel src -d lib -s
```

有时，我们也会以局部模块的方式安装 Babel-cli，这样的方式虽然会导致重复操作，但也减少了对全局环境的依赖，也能避免同一模块版本冲突问题。在这种方式下，我们通常结合 npm 使用，这需要在 npm 的配置文件 package.json 中添加一些片段。如下面的示例：

```
{
  // ...
  "devDependencies": {
    // …
    "babel-cli": "^6.0.0"
  },
  "scripts": {
    // …
"build": "babel src -d targetDir"
  },
}
```

执行转译就运行下面的命令。

```
$ npm run build
```

这样的好处是通过 npm 实现开发各环节的统一整合。

10.6.3 命令行运行工具：babel-node

和 babel-cli 工具一起发布的工具还有 babel-node，其以命令行方式提供直接运行 ES 6 代码的功能，可以直接输入代码也可以运行代码文件。它提供一个支持 ES 6 的 REPL 环境，支持 Node 的 REPL 环境的所有功能，而且可以直接运行 ES 6 代码。

babel-node 工具不用单独安装，是 babel-cli 自带的。因与主题关系不大，这里不再赘述，仅提示一点：babel-node 也可以结合 npm 使用，看下面的例子：

```
{
  "scripts": {
    "exampleModule": "babel-node example.js"
  }
}
```

运行 npm run exampleModule 可以直接运行 example.js 文件，且不作任何转译处理。

10.6.4 实时转译模块：babel-register

babel-register 模块用于在运行过程中进行实时转译，其原理是改写 require 命令，为之加上一个钩子，在钩子中进行转译处理。每次使用 require 加载指定的文件类型时，如带.js、.jsx 等后缀名的文件，就会自动启用 Babel 进行转译。安装命令如下：

```
$ npm install --save-dev babel-register
```

使用时，必须先加载 babel-register 模块，然后再加载其他模块。

```
require("babel-register");
require("./mymodule.js");
```

然后，基于 ES 6 规范编写的代码就会自动转译成 ES 5 代码。

基于 babel-register 的运行原理，babel-register 只会对使用 require 命令进行加载的文件进行转译。通常，实时转译在开发环节使用，发布时直接发布转译后的代码。

10.6.5 浏览器实时转译模块：browser.js

在浏览器前端也可以使用 Babel 进行转译，Babel 对应浏览器内实时转译的模块是 browser.js 文件，使用时将这个文件单独嵌入到 html 中即可。如下：

```
<script src="js/browser.js"></script>
<script type="text/babel">
```

```
// 这里编写 ES6 代码
</script>
```

需要注意的是：用 ES 6 代码编写的部分，其 script 标签的类型是 text/babel。

本书前面的章节为减少不必要的环节，均使用这种方式进行转译，但实际生产环境是不推荐这种方式的，因为转译的成本是很高的。本书后面的章节及示例均使用依托 npm 的完整开发环境。

Babel 从 6.0 版本以后都不再直接提供浏览器版本。我们前面使用的 browser.js 文件是从 5.x 版本的 babel-core 中复制出来的，实例包的示例中均包含这个文件。

10.6.6 转译API模块：babel-core

有些工具模块需要调用 Babel 的 API 进行转译，babel-core 模块就是这个用途，实际上 babel-cli、babel-register 等模块也是通过调用 babel-core 模块来完成转译的。这里不再展开介绍。

10.6.7 扩展转译模块：babel-polyfill

Babel 默认只对 ES 6 语法进行转译，而不考虑 ES 6 中新增的 API，如 Set、Maps、Iterator、Generator、Proxy、Reflect、Symbol、Promise 等对象或对象上新增的函数都不会转译。如果确实需要转译，可以使用 babel-polyfill 模块，这里不再展开介绍。

10.6.8 ESLint前置转译模块：babel-eslint

在使用 ESLint 进行代码语法和风格的静态检查时，可能也会有转译需求，babel-lint 可以解决这个问题：通常是在配置文件.eslint 中加入 parser 字段指明前置转译模块。如下：

```
{
  "parser": "babel-eslint",
  "rules": {
    ...
  }
}
```

然后就可以正常使用 ESLint 了。

10.6.9 Mocha前置转译模块：babel-core/register

Mocha 是常用的测试框架，在实际测试 React 组件时，如果使用了 ES 6 语法，也需要先对 ES 6 进行转译。Mocha 提供命令行参数--compilers 来指定转译模块。如：

```
mocha --compilers js:babel-core/register
```

在上面的命令中，--compilers 参数指定脚本的转译器，规定后缀名为 js 的文件，都需要使用 babel-core/register 先转译。

结合 npm 使用时需在 package.json 中增加 test 任务片段，如下：

```
"scripts": {
  // …
  "test": "mocha --compilers js:babel-core/register"
}
```

启动测试过程，只需要运行命令 npm run test 即可。

10.7 webpack打包工具使用与技巧

10.7.1 前端模块化与webpack介绍

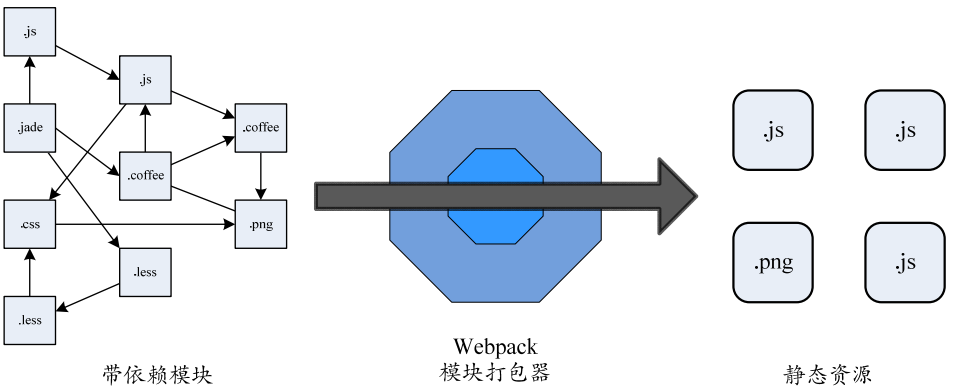
模块化的目的是使代码可以重用，模块化在任何开发中都是必不可少的。前端的模块化是 Node.js 带来的新产物，若干年前，前端模块化的需求并不是很强烈，因为在浏览器里运行的逻辑都比较简单。随着 Node.js 的出现，JavaScript 开始在后端发挥作用，为了代码重用，模块化就变得不可或缺。随着前端快速发展，前端业务越来越复杂，模块化的需求开始出现。

为了实现模块化，出现了用于 JavaScript 模块管理的两大流行规范：CommonJS 和 AMD。前者定义的是模块的同步加载，主要用于 Node.js。同步加载在前端会导致整个页面等待，对前端并不适用，便出现了 AMD。AMD 采用异步加载方式，通过 RequireJS 等工具适用于前端。以 RequireJS 为例，这是一种在线“编译”模块的方案，相当于在浏览器中先加载一个 AMD 解释器，使浏览器认识 define、exports、module 等相关命令，来实现模块化。后来 ES 6 提供了对模块的原生支持，它的目标是创建一种 CommonJS 和 AMD 使用者都愿意接受的方式，既拥有简洁的语法，又支持异步加载和配置模块加载。

而目前更通用的是 browserify、webpack 等技术，是一个预编译模块的方案。这个方案更加智能，因为它是在发布前预编译好的，不需要在浏览器中加载解释器。另外，直接写 AMD 或 ES 6 的模块化代码，它都能编译成浏览器识别的 JavaScript 代码。甚至 CommonJS 规范的模块化，browserify、webpack 也可以转换成浏览器使用的形式。webpack 相当于加强版 browserify，诞生伊始，瞄准的就是大型单页应用，而且其对 React 的支持是最好的，所以在本书中主要介绍 webpack。

webpack 是一个供浏览器环境使用的模块打包工具，其官方网址是 https://

webpack.github.io/。webpack 将项目中用到的一切静态资源都视为模块，模块之间可以互相依赖，webpack 对它们进行统一的管理及打包发布。下面引用 webpack 官方图片来说明 webpack 的作用。



从图中可以看出，webpack 的作用是对项目中的静态资源进行统一管理，为项目的发布提供最优的打包和部署方案，可以把应用中的 js、css、图片等资源集中打成一个或多个包文件。它具备编译打包 css、预处理 css、编译 js 和处理图片等许多功能，支持 AMD 和 CommonJS 规范，以及多种系统模块，现在已成为前端打包的主流工具。

webpack 的主要特性如下：

（1）同时支持 CommonJS 和 AMD 模块。

（2）串联式模块加载器及插件机制，具有更好的灵活性和扩展性，例如对 CoffeeScript、ES 6 的支持。

（3）可以基于配置或者智能分析打包成多个文件，实现分别满足公共模块和按需加载的需要。

（4）支持对 css、图片等资源进行打包，无须依靠 grunt 或 gulp 等构建工具，简化工程配置。

（5）开发时在内存中完成打包，性能更高，完全可以支持开发过程的实时打包需求。

（6）对 sourcemap 有很好的支持，易于调试。

10.7.2 webpack的打包React实例

这里将围绕 webpack 对 React 打包的相关用法做一个总体介绍。首先介绍 webpack 的安装和使用方法。

webpack 一般作为全局 npm 模块安装，安装命令如下：

```
$npm install -g webpack
```

安装之后便可以在命令行中直接使用 webpack 命令，直接执行此命令会默认使用当前目录的 webpack.config.js 作为配置文件。如果要指定另外的配置文件，可以执行：

```
$webpack --config <customConfigFile>
```

webpack 可以通过命令行来指定参数，在命令行中输入 webpack -h 可以查看有哪些参数。但是这种方式用起来不方便，在开发中通常会将所有相关参数定义在配置文件中。开发环境和生产环境一般会使用不同的打包方式，生产环境下的打包文件不需要包含 sourcemap 等用于开发的代码，而且需要尽可能地压缩和精简打包后的内容。因此，一般使用中会定义两个配置文件，一个用于开发环境，一个用于生产环境。配置文件通常放在项目根目录之下，其本身也是一个标准的 CommonJS 模块。

一个最简单的 webpack 配置文件 webpack.config.js 代码如下所示：

```
module.exports = {
  entry:[
    './app/main.js'
  ],
```

```
  output: {
    path: __dirname + '/build/',
    publicPath: "/build/",
    filename: 'bundle.js'
  }
};
```

其中，entry 参数定义了打包的入口文件，数组中的所有文件会按顺序打包。每个文件进行依赖的递归查找，直到所有依赖模块都被打包。output 参数定义了输出文件的位置，常用的参数包括：path（打包文件存放的绝对路径）、publicPath（网站运行时的访问路径）、filename（打包后的文件名）。

现在示范如何使用 webpack 打包一个 React 组件。假设有如下项目文件夹结构：

```
- react-sample
  + build/
   - js/
     Sample.js
     entry.js
   index.html
   webpack.config.js
```

在 Sample.js 定义了一个非常简单的 React 组件，代码如下：

```
import React from 'react';

class Sample extends React.Component {
  render() {
    return (
      <h1>Hello {this.props.name}!</h1>
    );
  }
}
module.exports = Sample;
```

entry.js 是入口文件，将一个 Hello 组件输出到界面：

```
import React from 'react';
import ReactDom from 'react-dom';
import Hello from './Sample';

ReactDom.render(<Hello name="React" />, document.body);
```

index.html 的代码如下，仅载入打包后的 bundle.js：

```
<html>
<head>
<script src="/build/bundle.js"></script>
</head>
<body>
</body>
</html>
```

由于 Sample.js 和 entry.js 都使用了 ES 6 和 JSX 语法，不能直接在浏览器中运行，因此需要对它们进行预处理转换。这项工作需要引入 webpack 的 Babel 加载器 babel-loader 和 Babel 对 JSX 和 ES 6 转换的预处理器。安装命令如下：

```
$npm install --save-dev babel-loader
$npm install --save-dev babel-preset-react
$npm install --save-dev babel-preset-es2015
```

在配置文件中加入如下配置：

```
module: {
    loaders: [{
        loader: "babel-loader",
        test: /\.jsx?$/,
        query: { presets: ['react', 'es2015']}
    }]
}
```

module.loaders 是 webpack 配置文件中最重要的内容，它告知 webpack 每一种

文件都需要使用什么加载器来处理。加载器一般命名为*-loader，有很多加载器可供使用，包括处理 CSS、ES 6、图片、URL 的众多加载器。本例中配置了 js 和 jsx 文件由 babel-loader 来处理；query 后面是 babel-loader 的参数，表示使用 React 预处理器，也可写成 babel-loader?presets[]=react,presets[]=es2015。

Babel 加载器可以将 JSX 和 ES 6 编译成 JavaScript，并加载为 webpack 模块。这样在当前目录执行 webpack 命令之后，在 build 目录将生成 bundle.js，打包了 entry.js 的内容。当浏览器打开当前服务器上的 index.html 时，将显示“Hello React!”。至此，使用 webpack 打包最简单的 React 工程介绍完毕，通过扩展这个工程，可以完成任意复杂的 React 打包工作。

在实际项目中，代码以模块进行组织，AMD 是在 CommonJS 的基础上考虑了浏览器的异步加载特性而产生的，可以让模块异步加载并保证执行顺序。而 CommonJS 的 require()函数则是同步加载。在 webpack 中推荐使用 CommonJS 的方式加载模块，这种方式的语法更加简洁直观。即使在开发时，也是加载 webpack 打包后的文件，然后通过加载 sourcemap 来进行调试。

除了项目本身的模块之外，对于需要依赖的第三方模块，可以使用 npm 进行管理，并使用 webpack 进行打包。例如，我们需要依赖 jquery，在命令行中执行：

```
$npm install jquery --save-dev
```

安装之后，在使用 jquery 的模块中需要在头部进行引入：

```
var $ = require('jquery');
$('body').html('Hello Webpack!');
```

本例中这种以 CommonJS 的同步形式引入其他模块的方式的代码更加简洁。执行中浏览器并不会实际地去同步加载这个模块，require 的处理是由 webpack 进行解析和打包的，浏览器只需要执行打包后的代码。webpack 本身已经可以完全处理 JavaScript 模块的加载，但是对于 React 中的 JSX 语法或 ES 6 语法，就需要额外引入 webpack 的加载器来处理了。

10.7.3 webpack模块加载器

webpack 将所有静态资源都视为模块，比如 JavaScript、CSS、LESS、JSX、CoffeeScript、图片等，从而可以对其进行统一管理。为此，webpack 引入了加载器（Loaders）的概念，每一种资源都可以通过对应的加载器处理成模块。前面已经介绍过利用 Babel 加载器处理 JSX 的例子。webpack 的加载器之间可以进行串联，一个加载器的输出可以成为另一个加载器的输入。比如，LESS 文件先通过 less-loader 处理成 CSS，然后再通过 css-loader 加载成 CSS 模块，最后由 style-loader 加载器对其做最后的处理，这样在运行时可以通过 style 标签将其应用到最终的浏览器环境。

为了让 webpack 识别资源应该用哪种加载器去载入，需要在配置文件中进行配置，即在 module.loaders 的 test 属性下编写正则表达式，通过正则表达式对文件名进行匹配，匹配成功则使用该加载器进行处理。例如：

```
module: {
    loaders: [{
        test: /\.less/,
        loader: 'style-loader!css-loader!less-loader'
    }, {
        test: /\.(css)$/,
        loader: 'style-loader!css-loader'
    }, {
        test: /\.(png|jpg)$/,
        loader: 'url-loader?limit=8192'
    }]
}
```

使用哪种加载器来处理文件完全取决于配置。加载器之间可以通过感叹号连接表示级联，例如对于 LESS 资源，写法为 style-loader!css-loader! less-loader。对于小型的图片资源，也可以将其进行统一打包，由 url-loader 实现，在代码 url-loader?limit=8192 中，limit 参数的含义就是对小于 limit 的图片资源使用 url-loader 进行打

包。图片打包这种方式，在一定程度上可以替代 CSS Sprites 方案，以有效降低前端对小图片的请求数量。

现在有很多第三方加载器可以实现常见静态资源的打包管理，基本上能满足日常开发需求，并不需要自行开发加载器，但对于特殊的需求，用户也可以开发自己的加载器。

10.7.4 webpack开发服务器

除了提供模块打包功能之外，webpack 还提供了一个基于 Node.js Express 框架的开发服务器（webpack-dev-server），这是一个静态资源 Web 服务器，简单静态页面或者仅依赖独立服务的前端页面，都可以直接使用这个开发服务器进行开发。在开发过程中，开发服务器会监听每一个文件的变化，进行实时打包，并且可以推送通知前端页面代码发生了变化，从而实现页面的自动刷新。

webpack 开发服务器需要单独安装，同样是通过 npm 进行，一般进行全局安装：

```
$npm install -g webpack-dev-server
```

之后便可以运行 webpack-dev-server 命令来启动开发服务器，然后再通过 localhost:8080/webpack-dev-server/访问到页面。默认情况下，服务器以当前目录作为服务器目录。在 React 开发中，通常会结合 react-hot-loader 来使用开发服务器。

10.7.5 React热加载器

webpack 运行时具有模块替换功能（Hot Module Replacement，HMR）。当某个模块代码发生变化时，webpack 实时打包将其推送到页面并进行替换，从而无须刷新页面就实现了代码的更新。如用户自行配置此功能，需要进行多方面考虑和配置，比较复杂，通用的做法是依赖第三方的 react-hot-loader 加载器，来实现 React 组件的热替换。其实正是因为 React 的每一次更新都是全局刷新的虚拟 DOM 机制，让 React 组件的热替换可以成为通用的加载器，从而极大地提高了开发效率。

要使用 react-hot-loader，首先需要通过 npm 进行安装：

```
$npm install --save-dev react-hot-loader
```

之后，需要在 webpack 开发服务器中开启 HMR 参数 hot。现在创建一个名为 server.js 的文件，用以启动 webpack 开发服务器，代码如下：

```
var webpack = require('webpack');
var WebpackDevServer = require('webpack-dev-server');
var config = require('../webpack.config');
new WebpackDevServer(webpack(config), {
  publicPath: config.output.publicPath,
  hot: true,
  noInfo: false,
  historyApiFallback: true
}).listen(3000, '127.0.0.1', function (err, result) {
  if (err) {
    console.log(err);
  }
  console.log('Listening at 127.0.0.1:3000');
});
```

为了热加载 React 组件，需要在前端页面中加入相应的代码，接收 webpack 推送过来的代码模块，并通知所有相关 React 组件进行重新渲染。代码如下：

```
entry: [
  'webpack-dev-server/client?http://127.0.0.1:3000',
  'webpack/hot/only-dev-server',
  './scripts/entry'
]
```

需要注意的是，这里的 client?http://127.0.0.1:3000 需要与 server.js 中启动 webpack 开发服务器的地址匹配。这样，打包生成的文件就知道该从哪里去获取动态的代码更新。然后，需要让 webpack 用 react-hot-loader 去加载 React 组件，加载器配置代码如下：

```
loaders: [
  {
    test: /\.js$/,
    exclude: /node_modules/,
    loader: 'react-hot!babel-loader?presets[]=react'
  },
  …
]
```

做完这些配置之后，使用 Node.js 启动 server.js：

```
$node server.js
```

现在即可启动开发服务器并实现 React 组件的热加载。为了方便，我们也可以在 package.json 的 scripts 中加入 start 配置：

```
"scripts": {
  "start": "node ./js/server.js"
}
```

从而通过 npm start 命令即可启动开发服务器。

现在任何的修改只要一保存，就会在页面上立刻体现出来。无论是对样式的修改，还是对界面渲染的修改，甚至是对事件绑定处理函数的修改，都可以立刻生效。

10.7.6 打包成多个资源文件

打包多个文件是 webpack 较之前的打包工具的主要进步之一。将模块打包成多个资源文件有两个目的：一是将多个页面的公用模块独立打包，从而可以利用浏览器缓存机制来提高页面加载效率；二是减少页面初次加载时间，只加载公共模块，其他模块可以动态按需去加载。

webpack 提供了非常强大的功能，让我们能够灵活地对打包方案进行配置。首先来看如何创建多个入口文件，代码如下：

```
{
  entry: {
    a: "./a",
    b: "./b"
  },
  output: {
    filename: "[name].js"
  },
  plugins: [
    new webpack.CommonsChunkPlugin("common.js")
  ]
}
```

可以看到，配置文件中定义了两个打包资源“a”和“b”，在输出文件中使用方括号来获得输出文件名。而在插件设置中使用了 CommonsChunkPlugin，在 webpack 中将打包后的文件都称为“Chunk”，这个插件可以将多个打包后的资源中的公共部分打包成单独的文件，这里指定公共文件输出到“common.js”。这样就获得了三个打包后的文件，在 HTML 页面中引用即可，代码如下：

```
<script src="common.js"></script>
<script src="a.js"></script>
<script src="b.js"></script>
```

除了在配置文件中对打包文件进行配置，还可以在代码中进行定义，例如：

```
require.ensure(["module-a", "module-b"], function(require) {
  var a = require("module-a");
  ...
});
```

webpack 在编译时会扫描到这样的代码，并对依赖模块进行自动打包，运行过程中执行到这段代码时会自动找到打包后的文件进行按需加载。

10.8 基于完整工具链的项目目录结构

下图展示了一个典型的 React 开发项目的目录结构。在实际开发时运行 npm start 即可启动一整套实时转译、打包的过程，并可以在浏览器中查看效果。只要在后台修改代码，浏览器中的界面也会自动实时更新。利用 npm 将 ESLint、babel、webpack、devServer 等系列工具组合在一起，可以达到“所写即所得”的效果，极大地提高了开发效率。

```
- example-testing
  - .babelrc
  - .eslintignore
  - .eslintrc
  - app
    - components
      - AddMessage.jsx
      - App.jsx
      - MessageItem.jsx
      - MessageList.jsx
    + img
    - stores
      - MessageStore.js
    - index.html
    - main.css
    - main.jsx
  + node_modules
  - package.json
  + test
  - webpack.config.js
  - webpack.production.config.js
```

项目根目录下包括各种配置文件，package.json 描述了整个工程包，包括基本信息和模块依赖关系等，.babelrc 为 babel 的配置文件，.eslintrc 文件为 ESLint 的配置文件等。webpack 有两个配置文件，webpack.config.js 对应开发时 webpack 的配置，webpack.production.config.js 对应实际部署时 webpack 的配置文件，运行 npm deploy 命令时会启用这个配置文件，并在 build 文件夹下存放最终文件。

在根目录下的文件夹中，app 存放所有的源代码，一般主 html、主入口 js 等文件放在 app 根目录下，其他的源代码文件按性质又划分为多个文件夹，便于管理。test 文件夹用于存放所有的测试代码。node_modules 是由 npm install 命令生成的模块文件存放的地方，有时候会比较大，我们一般不用进行处理。build 文件夹用于存放最后生成的部署用的文件。

第三篇

React进阶

一个前端应用的完整框架不仅要包含界面部分，还应该有界面切换管理和数据管理等部分。而作为一个优秀前端框架，React 的野心也不仅仅在于此。React 向完整的应用研发领域扩展，针对应用研发也试图给出一个完整的方案，其中支持数据管理的典型框架是 Flux 及 Redux，支持界面切换的框架则是 React-Router。本篇重点讲解这两个工程实用框架，通过这两个框架，进一步增强对数据和界面规范化管理的理解，更新前端开发理念。

除此之外，在实际工程开发中，单元测试是一个重要的环节。本篇最后重点介绍 React 单元测试的相关体系。其实可用来进行单元测试的技术路线并不止一条，但本书只选择了一条，即通过当前最主流的测试框架进行介绍。对于测试，我们并不太需要关注其原理，而是关注如何做好测试，因此本书的重点也围绕如何用好测试框架展开。

11

Flux & Redux

11.1 Flux

前面的章节我们所涉及的都是对 UI 的交互与控制，类似于 MVC 模式中的视图 View，但也不完全一样。如果是一个完整的应用，还应该包含处理数据模型的部分。当然我们也可以自行编写数据模型处理的部分，但其实数据模型处理的过程都有相似之处，会导致每次都编写出一些大同小异的代码，这样既不能提升代码质量又造成不必要的开销。为此，Facebook 提出了 Flux 应用架构来解决这个问题。

Flux 是一种思想，跟 React 本身没有必然联系，也可以用在其他框架上。我们在了解 Flux 的同时更要关注其架构思想和理念。

Flux 不仅是一个应用架构，更是一种架构思想。基于 Flux 思想的框架既有

Facebook 官方实现，也有 Redux、Reflux、Flummox 等产品，其中 Redux 就是其中的佼佼者。由于官方的 Flux 实现比较烦琐复杂，实际应用并不多，因此本书主要从原理上介绍 Flux，重点讲解在实际中应用较多的 Redux 框架。

将 Flux 用于 React 时，Flux 可以被看作从 UI 组件向前端应用整体的延伸。Flux 通过数据的单向流动为 React 可复用视图组件提供了扩展和补充。

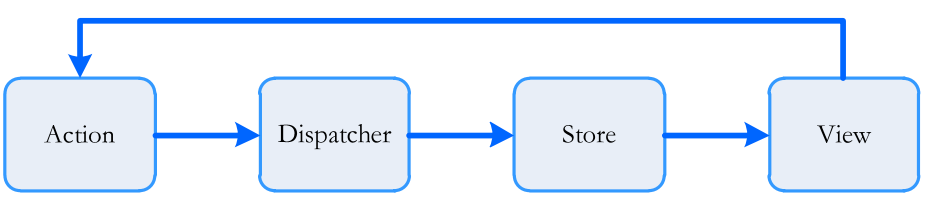
值得一提的是，使用 Flux 不需要也不应该更改现有的组件代码，因为 Flux 解决的是 UI 组件之外数据模型的问题，关注的是不同的领域。

11.1.1 Flux简介

Flux 作为一种应用的数据处理模式，是和 React 一起在 Facebook 成长起来的。事实上，在设计 React 之初就提出了 Flux 的思想，它们秉承同一个设计理念：单向数据流。

界面总是基于内部数据模型的一种呈现，但是用户在界面的输入也会回输到内部数据模型上，这是一种数据的双向流动。这种数据的双向传递流动很难把握，尤其是在复杂情况下，比如有些操作会触发一连串的变化，有些变化还可能是异步等，这样的数据流很难调试。

为此 Flux 引入了单向数据流的思想，当需要插入新的数据或改变原有数据时，数据流完全重新开始，如下图所示：



Flux 中的数据模型叫作存储（Store），Flux 中可以有多个 Store，里面存放着应用到的所有数据，类似于 MVC 中的 Model。视图 View 负责对 Store 中的数据进行

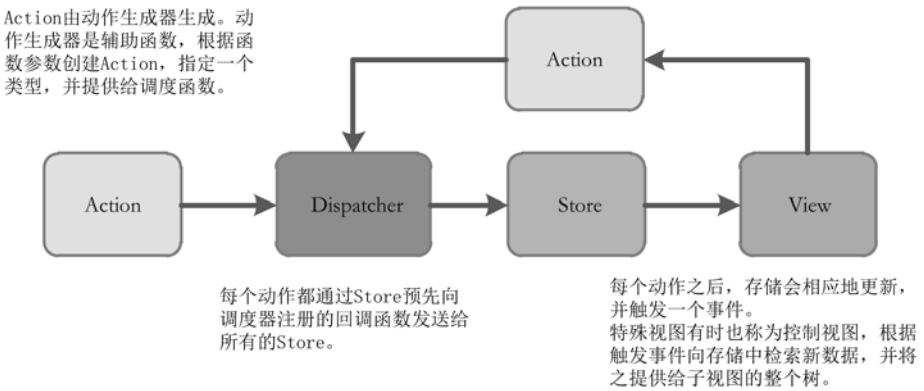
呈现，当 Store 发生变化时，应用的根组件（Controller View）会感知到变化，并根据最新的 Store 更新 View 中组件的 state，并最终将变化反映到视图上。

Flux 中 Store 的变化由动作 Action 激发，将动作 Action 理解为事件更合适。动作经由动作分发器 Dispatcher 派发到存储 Store 中，并引起 Store 中数据发生变更，从而影响视图的变化。一旦用户在视图中执行了某个操作，就会激发相应的 Action，上述过程就会重新开始一遍。

在 Flux 中看似是将一个简单的渲染过程复杂化，其实不然，一旦我们理解了 Flux 的架构，我们要做的事情就变得很简单了，只需要向框架中填入一些必要的代码即可。

11.1.2 基本架构

Flux 应用主要包括四部分：Dispatcher、Store、View 和 Action。Flux 的整个更新流程由 Action 驱动，Action 代表一个事件，通常由用户与 View 的交互产生。Action 由 Dispatcher 接收并转发，Dispatcher 是一个事件分发器，将接收到的 Action 交由 Store 来处理。Store 响应它们“感兴趣”的 Action，如果数据有变化就会触发一个 change 事件，来提醒 View 数据发生了变化，View 再重新从 Store 中获取最新数据，并重新渲染界面，完成整个更新过程并重新等待用户的交互。这里的 View 主要是指 React 组件，与 MVC 中的 View 并不完全等同。View 处于应用的最顶层，从 Store 中获取数据并进行展示，同时将这些数据传递给它的后代组件。当用户与 React 视图交互的时候，视图会产生一个 Action 对象记录操作的内容。这个 Action 会被 Store 接收并更新数据，View 从 Store 中重新获取数据并更新受影响的所有 View。这种方式与 React 组件的更新思路也是一脉相承的。



11.1.3 动作和动作发生器

Flux 中的动作（Action）是所有交互和改变的入口，改变应用的状态或有 View 需要更新时，都需要通过触发 Action 来实现。

Action 是一个 JavaScript Plain Object，主要记录对 State 的操作信息，如：

```
{
type : 'ADD_STUDENT',
name : '张三'
}
```

Action 对象包含两方面的内容：类型（type）和载荷（payload）。type 属性是通常预先定义好的常量 ID 标识符，代表特定的 action，是必要的。通常用有意义的变量名来描述操作 Store 的方式，比如上例中的 ADD_STUDENT。除 type 属性之外的其他属性统称为载荷，用于传递具体的数据，可以是任意格式。Action 的接收者负责解读和处理这个数据，只要接收者了解 Action 对象的结构就不会有问题。

Action 通常由动作发生器（Action Creator）创建，这样更便捷。一旦 Action 消息创建好了，Action Creator 就会直接将它注册到 Dispatcher。

11.1.4 分发器

分发器（Dispatcher）是 Flux 应用中的调度枢纽，管理着所有的数据流。Dispatcher 的主要职责是把 Action 分发给 Store。每个 Store 都会预先在 Dispatcher 中注册 Action 响应回调函数，当 Dispatcher 收到 Action 时就会调用每个 Store 所注册的回调函数。实际上，Dispatcher 维护着一个庞大的需要接收 Action 的 Store 回调函数列表，并接受注册。除此之外，Dispatcher 还可以扩展一些功能，如用来管理 Stores 之间的先后顺序依赖关系，这通过约定特定的顺序来调用回调函数就可以实现。

Dispatcher 把 Action 传递给所有登记在册的 Store，类似于广播，这与其他类似功能架构的 Dispatcher 有所不同。Store 也不是通过订阅某些 Action 来保持联系，而是聆听所有的 Action，并从中接收它所关心的 Action。

Dispatcher 通常实现为应用级的单例，也就是说，在应用程序中是唯一的。

Dispatcher 的行为是同步的，如果在 Store 之间有顺序关系，需要在某些 Store 更新后再更新，可以使用 Dispatcher 提供的 waitFor()函数来实现。

11.1.5 存储

存储（Store）负责封装应用的业务逻辑及和数据的交互，作用与传统 MVC 结构中的 model 相似。上面已经提到，Store 会在 Dispatcher 中注册一个回调函数来接收和处理 Action，这个回调函数接受 Action 作为参数，根据 Action type 识别 Action 并调用对应的处理函数，这个逻辑通过 switch 语句就可以完成。Action 的具体数据内容由 payload 参数携带。

某个 Action 的最终处理函数会根据 Action 所传递的 payload 更新 Store 中的 State，待 State 更新后，Store 会广播一个 change 更新事件来通知视图控制器状态已经发生变化了，然后 Views 就可以获取新的 State 并更新。

Store 在 Flux 应用中的地位是很重要的，它有以下几个特点：

- Store 中包含应用所有的数据。
- Store 是应用中唯一数据发生变更的地方。
- Store 中没有类似于 setState 这样的直接 setter 方法，而是通过向 Dispatcher 注册回调函数的方法实现数据的变更。因此要更新数据，必须通过规定的 Action Creator 和 Dispatcher 的方法实现。

11.1.6 视图与控制视图

视图（View）主要负责界面呈现并接受用户的输入，每一个 View 通常是一个 React 组件集合。整个 View 层是一个树形嵌套结构，这个嵌套结构的根部是一个特殊的 View，称为控制视图（Controller View）。

控制视图兼具控制功能，它一方面负责接收 Store 的广播事件，另一方面也从 Store 中获取数据（调用其 getter 函数），并且将数据传递到它的子代 View 中（调用组件的渲染函数）。

在实际使用中，有时候我们将整个 Store 作为属性传递到子组件中，让子组件自行选择所需要的数据，这样做的好处是避免了烦琐的属性逐层传递的过程，但也有副作用，对于复杂格式的数据而言，子组件必须了解传递进来的数据的结构，从而造成对外界不必要的依赖。

控制视图也可以不止一个。有时候在某一层次建立一个额外的控制视图能使逻辑得到简化。但非顶层的控制视图容易破坏单向数据流原则，导致数据流的冲突。这些都需要我们根据实际情况仔细权衡。

11.2 Redux

Facebook 提出了 Flux 思想来管理数据流，同时也给出了自己的一个实现，但这个实现既烦琐又难以使用，并没有得到广泛地推广使用。取而代之的是 GitHub 上涌现的一批针对 Flux 的实现框架，其中 Redux 因其简洁的模型和良好的编程体验正得到越来越多的应用。

Redux 与其他的 Flux 实现相比更简单，一个 Redux 应用中 Store 是唯一的，这样的限定能够使应用中的数据结果集中化，提高了可控性，大大地减少了开发和维护成本。同时又通过引入 Reducer 机制巧妙地解决单一 Store 的限制问题，达到了与使用多个 Store 同样的效果。

Redux 的使用可以不依赖于 React，它支持 Ember、AngularJS、jQuery 等多种框架，甚至还能只结合 JavaScript 使用。只不过我们大都结合 React 使用。

11.2.1 Redux基本架构

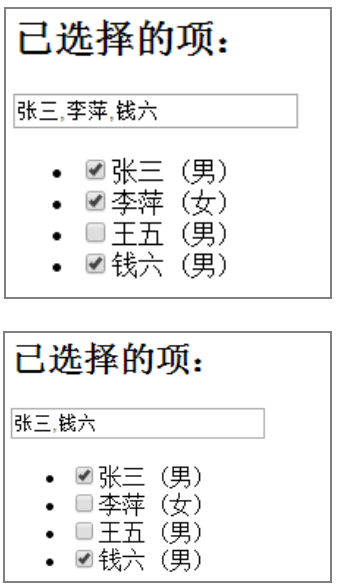
Redux 主要由 Action、Reducer 及 Store 三部分组成。Action 用来表达操作，Reducer 根据 Action 更新 State。

Redux 与 Flux 相比，主要有两点不同：

- Redux 中只有一个全局 Store 用来存储应用数据，Flux 里则可以有多个 Store。Flux 在 Store 里执行更新逻辑，当 Store 变化的时候再通知 Controller View 更新自己的数据。而 Redux 将各个 Store 整合成一个完整的 Store，而且 Redux 中的更新逻辑不在 Store 中执行，而是由 Reducer 处理。
- Redux 虽然也有 Dispatcher，但没有独立 Dispatcher 的概念，它将 Action 的创建和派发集中在一个函数中。事件的处理由 Reducer 负责，Reducer 根据

应用的状态和当前的 Action 生成新的 State。Redux 中可以有多个 Reducer，每个 Reducer 维护整体 State 中的一部分，多个 Reducer 合并成一个根 Reducer，由其负责维护完整的 State，当一个 Action 被发出，Store 会调用 Dispatch 方法向某个特定的 Reducer 传递该 Action，Reducer 接收到 Action 之后执行更新逻辑，并返回一个新的 State，根 Reducer 收集所有 State 的更新，再返回一个全新完整的 State，然后传递给 View。

下面结合一个选择器实例来讲解 Redux 原理，这个实例主要的用途是实现对多个用户的多项选择，界面运行效果如下。



图中的文本框用于显示当前已选中的用户，下部的列表项用于展示用户列表。每个用户项都带有一个 checkbox 项，选中该项，该用户名将自动加入到文本框中。取消选中该项则该用户名从文本框中消失。这是一个典型的多项选择器的应用案例。下面介绍其实现原理和代码。

11.2.2 Action

在 Redux 中，Action 的概念、使用和创建方法均与 Flux 中的一致，这里不再赘述。代码如下。注意：selectItem()函数所接收到的 info 参数描述了所选项和对所选项的操作（是否勾选），进而生成了不同的 Action。

```
    import { ADD_ITEM, DELETE_ITEM, DELETE_ALL, SELECTED_ADD_ITEM,
SELECTED_DEL_ITEM } from '../constants/actionTypes'

    export function addItem() {  return {type: ADD_ITEM};  }

    export function deleteItem(item) {
        return { type: DELETE_ITEM, item }
    }

    export function deleteAll() {
        return { type: DELETE_ALL }
    }

    export function selectItem(info) {
        if (info.checked) {
            return {
                type: SELECTED_ADD_ITEM,
                item: info.item
            }
        } else {
            return {
                type: SELECTED_DEL_ITEM,
                item: info.item
            }
        }
    }
```

11.2.3 Reducer

Reducer 的作用主要是根据 Action 执行更新 Store 中 state 的操作。Reducer 一般为简单的处理函数，传入参数为操作之前的 state 和指示操作的 action，返回新的 state。如下：

```
    import Immutable from 'immutable';
    import { SELECTED_ADD_ITEM, SELECTED_DEL_ITEM }
    from '../constants/actionTypes';
    const initialSelectedItems = Immutable.List();

    export    default    function    selectedItemsReducer(state    =
initialSelectedItems, action) {
        switch(action.type) {
            case SELECTED_ADD_ITEM:
                return state.push(action.item);
            case SELECTED_DEL_ITEM:
                return state.delete(state.indexOf(action.item),1);
            default:
                return state;
        }
    }
```

上面的代码根据传入 action 的 type 来识别对应的 action，并执行不同的 state 更新操作。

可以想象，当项目中存在越来越多的 action.type 时，上面的 Reducer 将变得越来越大，太多的 case 将导致代码难以维护。为了让代码结构更清晰，通常会将 Reducer 拆分成一个个小的 Reducer，每个 Reducer 分别处理 state 中的部分数据，最后再将处理后的数据合并成整个 state。

在这个实例中，我们将 state 分解为两个部分：items 和 selectedItems，items 对应待选用户列表，selectedItems 用于存储已选用户列表。上面代码展示的是对

selectedItems 的 Reducer，针对 items 的 Reducer 代码如下（在这个实例中实际上并没有真正用到，仅出于展示原理需要）：

```
    import Immutable from 'immutable';
    import    {    ADD_ITEM,    DELETE_ITEM,    DELETE_ALL    }    from
'../constants/actionTypes';

    const initialItems = Immutable.List([
        {name:'张三', gender:'男'},
        {name:'李萍', gender:'女'},
        {name:'王五', gender:'男'},
        {name:'钱六', gender:'男'}
    ]);

    export default function itemsReducer(state = initialItems, action)
{
        switch(action.type) {
            case ADD_ITEM:
                // 执行添加项的任务
                return state.push(action.item);
            case DELETE_ITEM:
                // 执行删除项的任务
                return state.delete(state.indexOf(action.item),1);
            case DELETE_ALL:
                return state.clear();
            default:
                return state;
        }
    }
```

最后，通过组合函数将上面两个 Reducers 组合起来，如：

```
    function rootReducer(state = {}, action) {
```

```
    return {
  items: itemsReducer(state.items, action),
  selectedItems: selectedItemsReducer(state.selectedItems, action)
    };
  }
```

上面的 rootReducer()将不同部分的 state 传给对应的 Reducer 处理，最终合并所有 Reducer 的返回值，组成整个 state。

实际上，Redux 提供了 combineReducers()函数来做 rootReducer 所做的事情。使用 combineReducers()来重构 rootReducer 的方法如下：

```
  import { combineReducers } from 'redux'
  import items from './items'
  import selectedItems from './selectedItems'

  const rootReducer = combineReducers({
    items: itemsReducer,
    selectedItems: selectedItemsReducer
  });
  export default rootReducer;
```

combineReducers()将调用一系列 Reducer，并根据对应的 key 筛选出 state 中的一部分数据给相应的 Reducer，这也意味着每一个小的 reducer 将只能处理 state 的一部分数据，如：itemsReducer()将只能处理及返回 state.item 的数据，如果需要使用其他 state 数据，就需要为这类 Reducer 传入整个 state。

在 Redux 中，一个 Action 可以触发多个 Reducer，一个 Reducer 中也可以包含多种 action.type 的处理，是多对多的关系。

11.2.4 Store

Redux 中的 Store 与 Flux 中的基本类似，但 Redux 中的 Store 是一个全局的 Store，

表现为一个单例。同时，Store 还兼具 Dispatcher 的功能，把 Action 和 Reducer 连接起来，根据 Action 来进行事件分发处理。Redux 中的 Store 是一个核心对象，相关的信息包括 state 数据、监听器、Reducer 等。

Store 提供了三个 API 来操作相关流程。其中：

- store.getState()用来获取 state 数据。
- store.subscribe(listener)用于注册监听函数。每当 state 数据更新时，将会触发监听函数。
- store.dispatch(action)用于将一个 action 对象发送给 Reducer 进行处理，如：

```
store. dispatch({
    type: 'ADD_ ITEM',
    name: '张三'
});
```

将 type 为 ADD_ ITEM 的 Action 发送到 Reducer 进行处理。通过调用 store.dispatch(action)的方式，使 Redux 能统一地对 state 进行管理。这种方式将数据的变化与数据的处理分离开来，形成松耦合的调用方式，使对数据的操纵更灵活，也便于代码组织和功能扩展。比如，我们想要增加 Action 的保存、删除、回滚、重置等功能，使用 Redux 就能比较轻松地实现。

Redux 中的 Store 需要由 createStore()函数创建，如：

```
var store = createStore(rootReducer,initialState);
```

这里的参数 rootReducer 为合并后的根 Reducer，initialState 为初始 state 数据。

为更清晰地理解 Store，下面给出了 createStore()函数的参考代码：

```
function createStore (reducer, initialState) {
    var currentReducer = reducer; //用于保存 reducer
    var currentState = initialState; //用于保存 state 数据
    var listeners = [ ] ; //用于保存跟踪 state 变化的监听器
```

```
    function getState() {
        return currentState;
    }

    function subscribe(listener) {
        listeners. push(listener) ;
        return function unsubscribe() {
            var index = listeners.indexOf(listener) ;
            listeners.splice(index, 1);
        };
    }

    function dispatch(action) {
        currentState = currentReducer(currentState, action);
        listeners.slice().forEach(listener = > listener());
        return action;
    }

    return {
        dispatch,
        subscribe,
        getState
    };
}
```

11.2.5 bindActionCreators

前面提到，我们通常使用 ActionCreator 来创建 Action。在实际调用 store.dispatch 时我们常常要重复写 store.dispatch(actionCreator(…)) 这样的代码，为避免这种无趣操作，我们可以对 store.dispatch 再进行一层封装，将两次调用转化为一次调用。Redux 提供的 bindActionCreators 就实现了这个转化。使用方式如下：

```
    var actionCreators = bindActionCreators(actionCreators, store.
dispatch) ;
```

经 bindActionCreators 包装后的 ActionCreator 变成了具有改变全局 state 数据的多个函数，可以很方便地通过调用这些函数来实现 state 的操作。

11.3 React-Redux

Redux 并不依赖 React 的存在而运行。因此，需要 React-Redux 作为桥梁实现 Redux 与 React 的连接。React-Redux 主要提供 Provider 和 connect 两个组件来建立 React 组件与 Store 中 state 数据之间的连接关系。前者确保 React 组件可被连接（connectable），后者把 React 组件和 Redux 的 Store 真正连接起来。

11.3.1 React-Redux的使用方法

使用 React-Redux 时，首先要创建一个 Provider 组件，作为最顶层的组件将所有 React 组件包裹起来，从而使所有的 React 组件都变为 Provider 的后代组件，再将已经创建好的 Store 作为属性传递给 Provider 组件。通过 Provider 组件建立起 Store 与 React 组件之间的联系。如下面代码所示：

```
    import React from 'react';
    import { render } from 'react-dom';
    import { createStore } from 'redux';
    import { Provider } from 'react-redux';

    import App from './containers/App';
    import rootReducer from './reducers';

    const store = createStore(rootReducer);
```

```
render(
    <div>
        <Provider store={store}>
            <App/>
        </Provider>
    </div>,
    document.getElementById('app')
);
```

Provider 内的任何一个组件（比如这里的 App），并不能直接获得 State 中的数据，而且也只能是有选择地访问 State 中的某些数据，这需要经过一个称为“connect”的过程。Connect 的主要作用是将 State 中的数据转换为组件可用的数据，以及生成 Action 的派发函数。在运行过程中 React-Redux 会调用 connect 对组件进行包裹，并传递 State 数据。下面是一个调用 connect()函数的示例：

```
export default connect(state => ({
    items: state.items,
    selectedItems: state. selectedItems
}), dispatch => ({
    actions: bindActionCreators(ItemsActions, dispatch)
}))(App)
```

如果不使用 Provider 和 connect 这样的机制，也可以在 React 中直接使用 Redux，在最外层容器组件中初始化 Store，然后将 State 上的属性作为 props 层层传递下去，但是这样做并不值得提倡。下面的例子是直接使用 Redux 的示范，与上例相比，我们能清楚地理解 React-Redux 的结构，关注点分离也更明晰。

```
class App extends Component{
  componentWillMount(){
    store.subscribe((state)=>this.setState(state))
  }
```

```
  render(){
    return <Comp state={this.state}
                 onIncrease={()=>store.dispatch(actions.increase())}
                 onDecrease={()=>store.dispatch(actions.decrease())}
    />
  }
}
```

11.3.2 Connect

Connect 主要把 State 和 Actions 转换为其下“木偶组件”所需要的 props。所谓木偶组件是指与 Redux 不产生直接联系的组件，如前面例子中 App 下的系列子组件，这些子组件既不知道 Store 的存在，也不知道 Action 的存在，只承担纯粹的组件职责，本书前面章节所介绍的组件基本上都可被看作木偶组件。

下面结合 App 来看看 Connect 的具体功能。

```
import React from 'react';
import { bindActionCreators } from 'redux';
import { connect } from 'react-redux';
import ImmutableRenderMixin from 'react-immutable-render-mixin';
import * as ItemsActions from '../actions';
import SelNames from '../components/selNames';
import UserList from '../components/userList';

let App = React.createClass({
   mixins: [ImmutableRenderMixin],
   propTypes: {
      items: React.PropTypes.object,
      selectedItems: React.PropTypes.object
   },
   render() {
```

```
        const actions = this.props.actions;
        const items = this.props.items;
        const selectedItems = this.props.selectedItems.toArray();

        return (
            <div>
                <h2>已选择的项：</h2>
                <SelNames selItems={selectedItems}/>
                <UserList items={items} selectItem={actions. selectItem}/>
            </div>
        )
    }
})

export default connect(state => ({
    items: state.items,
    selectedItems: state.selectedItems
}), dispatch => ({
    actions: bindActionCreators(ItemsActions, dispatch)
}))(App)
```

在上面的例子中，Connect 和 Provider 组件相配合，将 actions、items、selectedItems 以 props 的形式传入到 App 木偶组件中，使 App 组件能直接使用 this.props.actions、this.props.items 和 this.props.filters。Connect 其实就是一个高阶组件，将木偶组件进行高一层的包裹，以便传递 state 中的数据。

木偶组件的事件被激发会引起 Actions 方法被调用，Actions 中的函数是经过 bindActionCreators 加工处理过的，所以形如 actions.addItem 会直接创建动作并进行派发，从而驱动 store.state 更新，并触发 Store 中预先通过 subscribe 注册的回调函数，最终引起视图的重新渲染。更新时使用新的数据重新渲染组件的过程，实际上是为木偶组件传入了新的 props，引发一级的差异比较和 DOM 更新，最终体现出来的效果就是页面更新了。

下面详细介绍 connect()的调用参数。

```
connect([mapStateToProps], [mapDispatchToProps], [mergeProps], [options])
```

connect() 接收四个参数，分别是 mapStateToProps、mapDispatchToProps、mergeProps 和 options。

```
mapStateToProps(state, ownProps)
```

这个函数定义将 Store 中的哪些数据作为 props 传递到 React 组件上，返回值就是要传递到 React 组件的数据。

这个函数本身接收的第一个参数就是 Store 中的 state，我们从中选取数据并合并到返回值中。通常我们没有必要将 state 中的数据原封不动地传入组件，而是根据 state 中的数据，有选择地输出组件所需要的属性。

这个函数接收的第二个参数是 ownProps，是组件自身的 props，这个参数是可选的。有的时候，组件自身的 props 也会有一些影响，因为组件的 props 并不完全来自于 state，有些是用户预先定义的固有属性。

当 state 的 ownProps 发生变化的时候，mapStateToProps()函数都会被调用，得到的新 stateProps 会传递给 React 组件。

```
mapDispatchToProps(dispatch, ownProps)
```

这个函数的作用是将 Actions 作为 props 绑定到 React 组件上，函数的返回结果也会被合并到组件的 props 中。

为了不让用户感知到有 dispatch 的存在，我们通常将用户定义的 action()函数和 dispatch()函数合并成一个新函数，这个调用 Redux 本身提供的 bindActionCreators()函数来完成。

其接收的第二个参数 ownProps 的含义和作用与 mapStateToProps()函数一样。当 ownProps 变化的时候，该函数也会被调用，生成的新 dispatchProps 再传递给

React 组件。但有一点需要注意，Action 变化不会引发 mapDispatchToProps()被调用，因为 Actions 集合在组件生命周期中是固定的。

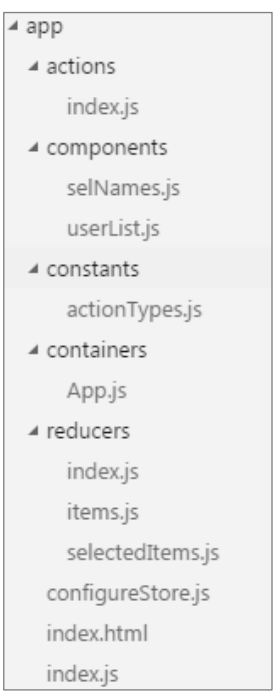
```
mergeProps(stateProps, dispatchProps, ownProps): props
```

前面两个函数调用后的结果都需要与 ownProps 进行合并后才会传递给 props，具体的合并方式由 connect() 的第三个参数 mergeProps()函数负责。一般情况下，没有必要传递这个参数，默认会使用 Object.assign()函数来完成合并工作。

最后的 options 选项一般不会用到，这里略过不提，有兴趣的读者可以自行学习。

11.4 Redux工程目录结构

这里给出前面实例的完整工程目录结构。这个实例在书中所附实例包中的 chapter11/example-redux 文件夹。要查看效果，请将该文件夹复制至硬盘下，并在新文件夹下以命令行界面运行 npm install（如已有 node_modules 文件夹则可省略这一步），完成后再运行 npm start 即可。



完整的目录结构如右图所示。使用 npm start 可以启动整个工程。

1．index.js

index.js 文件是总的入口文件，在这个文件中建立了 App 和 Redux 之间的联系。App 组件是应用的顶层组件，并作为 Provider 组件的直接子组件渲染。Provider 组件把 Store 和视图（顶层的 App 组件）绑定在了一起，这里的 Store 中

包含了那个唯一的 State 树。当 Store 发生改变的时候，整个 App 就可以作出对应的变化。早期的 React 需要将 App 声明为一个函数，现已不再采用，而是直接声明为组件即可。

2．containers/App.js

App.js 主要描述 App 组件并调用 connect()函数，使 App 组件能接收到 actions、items、filter 数据。其中 this.props.actions 中保存的是各个 Action 驱动函数，如 actions.addItem 等，App 组件在这里相当于一个总的视图，也是一个总的装配器。

3．constants/actionTypes.js

actionTypes.js 主要包含对 Action 中 type 的声明，每个 type 就是一个常量。

```
import keyMirror from 'fbjs/lib/keyMirror'

export default keyMirror({
    ADD_ITEM: null,
    DELETE_ITEM: null,
    DELETE_ALL: null,
    SELECTED_ADD_ITEM: null,
    SELECTED_DEL_ITEM: null
})
```

这里使用了 keyMirror 工具，它的主要用途就是创建与键值 key 相等的常量。上面的例子等价于：

```
export const ADD_ITEM = 'ADD_ITEM';
export const DELETE_ITEM = 'DELETE_ITEM';
export const DELETE_ALL = 'DELETE_ALL';
export const SELECTED_ADD_ITEM = ' SELECTED_ADD_ITEM ';
export const SELECTED_ DEL_ITEM = ' SELECTED_ DEL_ITEM ';
```

4．actions/index.js

这个文件主要包含各个 Action 驱动函数，每个函数都响应用户事件，创建 Action 并向 store 派发。最终的响应函数是基于 Action 驱动函数，并经由 bindActionCreators 加工过的。在这些函数中，还可以进行异步请求，如 ajax 等。

5．reducers/items.js

该文件包含对 items 数据进行增、删、改操作的处理。要注意 Action 操作携带的参数。

6．reducers/index.js

该文件把所有的 reducer 合并为一个根 rootReducer，需要调用 combineReducers() 函数来完成。利用这个机制就把 Reducers 拆分成一个个小的 Reducer 来管理 Store 了。

7．components

在 components 目录下的各组件是木偶组件，具体负责界面的展示和与用户交互。这些组件只承担它应用的职能，不知道也不关心 Actions 和 Stores 的存在，这些组件在各项目中是可共用的。

其中，SelNames 组件提供显示所选用户名的功能，UserList 组件负责用户列表的显示并提供勾选框，以选中或取消用户。

详细代码可参看随本书发布的范例，这里不再赘述。

12

路 由

12.1 前端路由

当前的 Web 应用开发，越来越倾向于单页应用（SPA）方式，而在单页应用中路由系统往往是必不可少的一部分。路由系统的主要作用是在浏览器当前的 URL 发生变化时做出对应的响应，用来保证用户界面与 URL 的同步。在传统的多页面应用开发中，路由的概念仅仅存在于后端 Web 应用服务器端（后端路由），现在则逐渐向前端延伸，形成前端路由框架。从性能和用户体验的角度来看，后端路由每次访问一个新页面的时候都要向服务器发送请求，然后服务器响应请求；而前端路由访问一个新页面的时候仅仅变换了一下路径，没有网络延迟。因此前端路由相较于后端路由，在性能和用户体验上会有相当大的提升。

目前的主流前端框架 Ember、Backbone、AngularJS、React 等都有自己的路由系统。React-Router 则是与 React 配套的前端框架定制开发的路由系统，也是 React 系官方维护的唯一路由库，它通过管理 URL，实现 React 组件的切换和状态的变化。

12.2 路由的基本原理

路由的作用虽然很重要，但其原理其实并不复杂，总的来说是保证视图和 URL 的同步，可以把视图看成资源的一种表现。当用户在页面中进行操作时，应用会在若干个交互状态中切换，比如用户会单击浏览器的前进、后退按钮进行切换等。路由系统的职责之一就是记录一些重要的状态，比如用户的登录状态、当前访问资源、用户的上一访问资源等，并根据需要重新以同步或异步的方式向服务端请求获取资源，然后重新渲染视图。

12.3 安装与引用

React-Router 作为一个 JavaScript 模块，使用前需先进行安装，安装命令如下：

```
npm install react-router
```

然后在代码中引用：

```
import { Router } from 'react-router';
```

12.4 路由配置

12.4.1 路由器和路由

React-Router 的路由功能主要由路由器（Router）和路由（Route）两个组件配合完成。路由器本身就是 React 组件，它们的使用方法没什么区别，但与普通 React 组件不同的是，路由器通常表现为一个容器。容器中具体呈现的内容，也就是路由，要通过 Route 组件定义，并在这些 Route 组件之间切换。下面的代码展示了两者的配合关系：

```
import { Router, Route, hashHistory } from 'react-router';
render((
  <Router history={hashHistory}>
    <Route path="/" component={App}/>
  </Router>
), document.getElementById('app'));
```

在上面的代码中，用户访问根路由/（如 http://www.example.com/），组件 App 就会被加载到 document.getElementById('app')。

这里的 Router 组件有一个参数 history，并被赋值为 hashHistory，表示路由的切换由 URL 的 hash 变化决定，即 URL 的#部分发生变化。举例来说，如果用户访问 http://www.example.com/，实际上看到的是 http://www.example. com/#/。

Route 组件定义了 URL 路径与组件的对应关系,这个对应关系通过对 path 属性和 component 属性的赋值声明。我们可以同时使用多个 Route 组件。

```
<Router history={hashHistory}>
  <Route path="/" component={App}/>
```

```
    <Route path="/help" component={Help}/>
    <Route path="/about" component={About}/>
  </Router>
```

在上面的代码中，用户访问/help（如 http://localhost:8080/#/help）时，加载 Help 组件；访问/about（http://localhost:8080/#/about）时，加载 About 组件。

12.4.2 嵌套路由

Route 组件可以嵌套使用，如下面的代码所示，用户访问/index 时，会先加载 App 组件，然后在它的内部再加载 Index 组件。

```
<Router history={hashHistory}>
  <Route path="/" component={App}>
    <Route path="/index" component={Index}/>
    <Route path="/about" component={About}/>
  </Route>
</Router>
```

在 App 组件的写法中，要用 this.props.children 属性代表子组件。

```
export default React.createClass({
  render() {
    return <div>
      {this.props.children}
    </div>
  }
})
```

子路由也可以不写在 Router 组件里，而是作为 Router 组件的 routes 属性传入。

```
let routes = <Route path="/" component={App}>
  <Route path="/repos" component={Repos}/>
  <Route path="/about" component={About}/>
```

```
</Route>;
<Router routes={routes} history={browserHistory}/>
```

12.4.3 默认路由

对于如下的路由设置，当用户访问根路径“/”时，App 组件本身会加载，但不会包含任何子组件导致出现大片空白，因为 App 组件的 this.props.children 此时为 undefined。

```
<Router >
  <Route path="/" component={App}>
    <Route path="/index" component={Index}/>
    <Route path="/about" component={About}/>
  </Route>
</Router>
```

为解决此问题，Router 提供了默认路由（IndexRoute）组件，用来显式指定根路由的子组件，即指定默认情况下加载的子组件。采用 IndexRoute 后的代码如下：

```
<Router>
  <Route path="/" component={App}>
    <IndexRoute component={Home}/>
<Route path="/index" component={Index}/>
    <Route path="/about" component={About}/>
  </Route>
</Router>
```

现在，用户访问“/”的时候，默认的 Home 组件作为 App 的子组件被加载，此时组件结构如下：

```
<App>
  <Home/>
</App>
```

使用 IndexRoute 组件，使结构清晰且统一，App 组件只包含下级组件的共有元素，本身的默认展示内容则由 Home 组件定义。

注意，IndexRoute 组件没有也不需要路径参数 path。

12.4.4 path 属性

Route 组件的 path 属性指定路由的匹配规则。一旦路由匹配，则对应的组件被加载。path 属性不以“/”开头时使用的是相对路径。请看下面的例子。

```
<Route path="inbox" component={Inbox}>
  <Route path="messages" component={Messages} />
</Route>
```

在上面的代码中，当用户访问/inbox/messages 时，会加载下面的组件。

```
<Inbox>
  <Messages/>
</Inbox>
```

嵌套的路由如果想摆脱这个规则，可以使用绝对路由，即以“/”开头。

Route 组件的 path 属性是可选的，但如果省略则认为路径始终匹配，即总是会加载对应的组件。下面的例子省略了外层 Route 的 path 参数，但将 inbox 路径移到里层的 Route 组件上，效果与上例相同。

```
<Route component={Inbox}>
  <Route path="inbox/messages" component={Messages} />
</Route>
```

path 属性支持使用通配符，下表列出了通配符的匹配规则。如果多个规则匹配，则只执行最先遇到的规则，执行规则的先后顺序即从上往下出现的顺序。

通配符规则	功　能	示　例

:paramName	匹配 URL 的一个部分，直到遇到下一个/、?、#为止。这个路径参数可以通过 this.props.params.paramName 取出	<Route path="/hello/:name">
		// 匹配 /hello/michael
		// 匹配 /hello/ryan

()	URL 的部分是可选的	<Route path="/hello(/:name)">
		// 匹配 /hello
		// 匹配 /hello/michael

*	匹配任意字符，直到模式里面的下一个字符为止	<Route path="/files/*.*">
		// 匹配 /files/hello.jpg
		// 匹配 /files/hello.html

**	匹配任意字符，直到下一个/、?、#为止	<Route path="/**/*.jpg">
		//匹配 /files/hello.jpg
		//匹配 /files/path/to/file.jpg

路由匹配规则是从上到下执行的，一旦发现了匹配的规则，就不再考虑其他的规则了。设置路径参数时，需要特别注意这一点。如这个例子：

```
<Router>
  <Route path="/:userName/:id" component={UserPage}/>
  <Route path="/about/me" component={About}/>
</Router>
```

在上面的代码中，用户访问/about/me 时，不会触发第二个路由规则，因为它会匹配/:userName/:id 这个规则。因此，带参数的路径通常要写在路由规则的底部。

在进行通配符匹配时，如何获得匹配的变量值呢？比如获得 URL 字符串/foo?bar=baz 中 foo 后面的参数值。Route 会将参数通过 this.props.location.query 传入组件，因此可以通过 this.props.location.query.bar 获取属性值。

12.4.5　NotFoundRoute组件

NotFoundRoute 组件用于声明父组件匹配成功但没有找到匹配的子组件时所激

活的子组件。通常利用它来处理不合法的链接。

NotFoundRoute 组件的 handler 属性用来声明处理路由不匹配的子组件。使用方法如下：

```
    <Route path="/" handler={App}>
      <Route name="course" path="course/:courseId" handler={Course}>
        <Route   name="course-dashboard"   path="dashboard"   handler=
{Dashboard}/>
        <NotFoundRoute handler={CourseRouteNotFound} />
      </Route>
      <NotFoundRoute handler={NotFound} />
    </Route>
```

在这个例子中，访问/course/123/×××将会激活 CourseRouteNotFound 组件，且渲染在 Course 内。而访问/×××则会激活 NotFound 组件。

NotFoundRoute 并不是针对资源没有被找到而设计的。路由没有匹配到特定的 URL 与通过一个合法的 URL 没有查找到资源是有区别的。如 course/123 是一个合法的 URL 并能够匹配到对应的路由，但是却没有匹配的组件。

12.4.6 Redirect组件

Redirect 组件用于路由的跳转，即用户访问一个路由，会自动跳转到另一个路由。

```
    <Route path="inbox" component={Inbox}>
      {/* 从 /inbox/messages/:id 跳转到 /messages/:id */}
      ＜Redirect from="messages/:id" to="/messages/:id" />
    </Route>
```

在这个例子中，访问/inbox/messages/5 会自动跳转到/messages/5。

12.4.7 IndexRedirect 组件

IndexRedirect 组件用于访问根路由的时候，将用户重定向到某个子组件。

```
<Route path="/" component={App}>
  ＜IndexRedirect to="/welcome" />
  <Route path="welcome" component={Welcome} />
  <Route path="about" component={About} />
</Route>
```

在上面的代码中，用户访问根路径时，将自动跳转到/welcome，最终激活了 Welcome 子组件。

12.4.8 history属性

Router 组件的 history 属性，用来监听浏览器地址栏的变化，并将 URL 解析成一个地址对象，供 React Router 匹配。

可以将 history 属性设置为 browserHistory、hashHistory 或 createMemoryHistory 三者之一，取决于具体需要，设置为 browserHistory 时居多。

当 history 属性设置为 hashHistory 时，路由会计算 URL 的 hash 值，并依据这个 hash 值进行切换，URL 形如/index.html#/route1?_k=soe6f7。

```
import { hashHistory } from 'react-router'

render(
  <Router history={hashHistory} routes={routes} />,
  document.getElementById('app')
)
```

如果 history 的属性设为 browserHistory，浏览器的路由显示正常的路径/index.html/route1，其底层是依托于浏览器本身的 History API 实现的。

```
import { browserHistory } from 'react-router'

render(
  <Router history={browserHistory} routes={routes} />,
  document.getElementById('app')
)
```

browserHistory 这种方式是我们经常使用的，但其也存在一定的问题。由于地址是在本地切换完成的，而不是由服务器响应的，如果向服务器请求子路由则会显示网页找不到的 404 错误。如果使用 webpack-dev-server 作为开发服务器，在命令行加上--history-api-fallback 选项可以避免这个问题。

```
$ webpack-dev-server --inline --content-base . --history-api-fallback
```

createMemoryHistory 方式主要用于服务器渲染。它只是创建一个内存中的 history 对象，并不与浏览器 URL 互动。因为这种方式不经常使用，所以这里不详细展开。

12.4.9 路由回调

每个路由都有 Enter 和 Leave 两个回调函数，在用户进入或离开该路由时被触发。

```
<Route path="about" component={About} />
＜Route path="help" component={Help}>
  ＜Redirect from="messages/:id" to="/messages/:id" />
</Route>
```

在上面的代码中，如果用户离开/messages/:id 进入/about，会依次触发以下函数。

```
/messages/:id 的 onLeave
/inbox 的 onLeave
/about 的 onEnter
```

下面的这个例子，使用 onEnter 回调函数实现了与 Redirect 相同的功能。

```
<Route path="inbox" component={Inbox}>
  <Route
    path="messages/:id"
    onEnter={
      ({params}, replace) => replace(`/messages/${params.id}`)
    }
  />
</Route>
```

利用 onEnter 函数还可以实现认证功能。

```
const requireAuth = (nextState, replace) => {
    if (!auth.isAdmin()) {
        // 如果不是管理员则跳转到主页
        replace({ pathname: '/' })
    }
}
export const AdminRoutes = () => {
  return (
     <Route path="/admin" component={Admin} onEnter={requireAuth} />
  )
}
```

12.5 路由切换

前面主要介绍了路由配置，那么在实际运行时，如何激活路由的切换动作呢？对于设计阶段就已经明确的路由切换，通过声明相应的路由切换组件即可完成，包括 Link 组件、IndexLink 组件等；而运行时的路由切换，则需要通过调用路由 API 来完成。

12.5.1 Link组件

Link 组件用于取代<a>元素，生成一个链接，允许用户单击后跳转到另一个路由。它基本上就是<a>元素的 React 版本，可以接收 Router 的状态。

```
render() {
  return <div>
    <ul role="nav">
      <li><Link to="/about">About</Link></li>
      <li><Link to="/repos">Repos</Link></li>
    </ul>
  </div>
}
```

如果希望当前的路由与其他路由有不同的样式，可以使用 Link 组件的 activeStyle 属性。

```
<Link to="/about" activeStyle={{color: 'red'}}>About</Link>
<Link to="/repos" activeStyle={{color: 'red'}}>Repos</Link>
```

按照上面的代码进行设置，当前页面的链接会红色显示。

另一种做法是，使用 activeClassName 指定当前路由的 Class。

```
<Link to="/about" activeClassName="active">About</Link>
<Link to="/repos" activeClassName="active">Repos</Link>
```

按照上面的代码进行设置，当前页面的链接的 Class 会包含 active。

12.5.2 IndexLink

如果链接到根路由/，则不能使用 Link 组件，而要使用 IndexLink 组件。因为根路由的特殊性，/会匹配任何子路由。使用 IndexLink 组件可以对路径进行精确匹配。

```
<IndexLink to="/" activeClassName="active">
```

```
    Home
  </IndexLink>
```

在上面的代码中，根路由只会在精确匹配时才具有 activeClassName 属性。

如果不想使用 IndexLink 组件，也可以使用 Link 组件中的 onlyActiveOnIndex 属性达到同样的效果。

```
  <Link to="/" activeClassName="active" onlyActiveOnIndex={true}>
    Home
  </Link>
```

实际上，IndexLink 组件就是对 Link 组件的 onlyActiveOnIndex 属性封装后的高阶组件。

12.5.3 动态路由切换

Link 组件用于正常的用户单击跳转，是预先声明式的。但是有时还需要作一些动态的路由切换，如表单跳转、单击按钮跳转等操作。这些情况只能通过 API 调用完成。

第一种方法是使用 browserHistory.push()函数，这种方法需要预先将 Router 的 history 属性声明为 browserHistory。

```
import { Router, Route, browserHistory } from 'react-router';
render((
  <Router history={ browserHistory }>
    <Route path="/" component={App}/>
  </Router>
), document.getElementById('app'));
```

然后在相应的动作响应函数中调用 browserHistory.push()函数：

```
  doRouteSwitch() {
    const path ='/repos/${userName}/${repo}';
```

```
      browserHistory.push(path);
    },
```

但第一种方法现在已不提倡使用，取而代之的是第二种方法，利用 context 机制传递 router 对象，在 Router 中声明的组件在 this.props.context 属性中都会包含 router 对象，从而调用 router.push(path)函数完成路由切换。

```
export default React.createClass({
  // 从 context 中取得 router 对象
  contextTypes: {
    router: React.PropTypes.object
  },

  doRouteSwitch(event) {
    // ...
    this.context.router.push(path);
  },
})
```

13

React单元测试

现在，页面前端逻辑的复杂度与日俱增，单元测试是在开发周期中应对复杂性的一个必不可少的重要环节。引入单元测试能显著降低维护 JavaScript 代码时的出错风险，保证代码质量，更重要的是减少人力测试过程，降低代码维护成本。

React 作为一个完整的解决方案，也离不开单元测试的配套工具。React 具有的虚拟 DOM、面向组件化的设计和 JSX 语法等特性，使其无法使用传统的测试工具，需要引入新的测试方法。

React 单元测试工具也先后出现了 Jasmine、Karma、Mocha、Chai、Sinon、Vows.js 和 QUnit 等众多测试工具。对于测试工具，我们并不关心各工具之间的区别，掌握其使用方法即可。为此，本书主要介绍目前主流的测试框架 Mocha，配合使用断言 Chai 等工具，主要列举一些经常用到的方法，不作详细深入探讨。

13.1 测试脚本示例

先从一段最简单的、与 React 无关的测试脚本着手。

```
import { expect } from 'chai';
import add from '../app/utils/add'

describe('最简测试', function() {
  it('1 加 1 应该等于 2', function() {
    expect(add(1, 1)).to.be.equal(2);
  });
});
```

上面的测试脚本实现了对 app/utils 文件夹下 add 模块中 add()函数的功能测试。

从上面的示例可以看到测试脚本的基本结构。测试脚本中包括一个或多个 describe 函数块，一个 describe 函数块称为一个“测试套件（test suite）”，包含一组相关的测试。describe()函数有两个参数，第一个参数是测试套件的名称（最简测试），第二个参数是实际要执行的测试函数。describe 函数块还可以嵌套。

每个 describe 函数块包括一个或多个 it 函数块。一个 it 函数块称为一个“测试用例（test case）”，对应一项独立的测试，是测试的最小单位。it()函数也有两个参数，第一个参数是测试用例的名称（1 加 1 应该等于 2），第二个参数是实际执行该项测试的函数。

it 函数块中的测试代码实现对功能的具体测试，只要测试代码运行不出错就认为测试通过。我们通常在测试代码中实现对功能的验证，通过断言函数 expect()进行验证。在上面的代码中，我们断言 add()函数的结果是“2”，如果结果不是“2”，expect 断言代码将会抛出异常，从而使这项测试失败。

这段测试使用了一系列的工具。测试框架采用的是 Mocha（支持 describe()函数和 it()函数），断言函数来自于 chai 模块（提供 expect 系列函数），这些工具我们在后面会进行深入介绍。接下来看看针对 React 的测试代码示例。

13.2 React 测试代码示例

下面的代码使用 React 官方的测试模块实现对 React 组件的功能测试。

```
    import React from 'react';
    import TestUtils from 'react-addons-test-utils';
    import { expect } from 'chai';
    import App from '../app/components/App';

    describe('React App 组件测试', function () {
      it('测试 todoItem 删除功能', function () {
        const app = TestUtils.renderIntoDocument(<App/>);
        let todoItems = TestUtils.scryRenderedDOMComponentsWithTag(app,
'li');
        let todoLength = todoItems.length;
        let deleteButton = todoItems[0].querySelector('button');
        TestUtils.Simulate.click(deleteButton);
     let                    todoItemsAfterClick                    =
TestUtils.scryRenderedDOMComponentsWithTag(app, 'li');
        expect(todoItemsAfterClick.length).to.equal(todoLength - 1);
      });
    });
```

上面的脚本实现了对 App 组件功能的测试。测试先引入 React 附带的测试专用模块“react-addons-test-utils”，该模块可以模拟 React 的渲染过程、单击事件等。DOM 渲染测试则使用了 jsdom。下面具体介绍测试时涉及的一系列工具。

13.3 React 测试相关工具

13.3.1 Mocha

Mocha 是现在的主流 JavaScript 测试框架之一，其优势是支持多种类型的断言 libs，支持异步和同步测试、支持多种方式的结果导出。与 Mocha 类似的测试框架还有 Jasmine、Karma、Tape 等，本书不涉及这些测试框架，读者可以搜索相关资料自行学习。

Mocha 是一个命令行工具，使用前需要运行 npm install –g mocha 命令进行安装，安装完成后可以直接在命令行运行 mocha 命令。

通常，测试脚本与所要测试的源码脚本同名，对应后缀名为.test.js 或.spec.js。比如，app.js 对应的测试脚本名是 app.test.js。

如果测试脚本使用 ES 6 语法，那么在运行测试之前，需要使用 Babel 进行转码。然后，在项目目录下新建一个.babelrc 配置文件。

```
{
  "presets": [ "es2015" ]
}
```

最后，使用--compilers 参数指定测试脚本的转译器。命令如下：

```
$ mocha --compilers js:babel-core/register
```

在上面的命令中，--compilers 参数后面的字符串指定了所用的转译器，转译器以“:”进行分隔，左边是要转译文件的后缀名，右边是要处理的文件的模块名。在运行测试之前，先使用 babel-core/register 模块处理所有后缀名为.js 的文件。Mocha 也可以在项目内安装（不使用-g 选项），如果转译器是安装在项目内的，那

么就要使用项目内安装的 Mocha；如果转译器是安装在全局的，则应该使用全局的 Mocha。

在异步调用场景中，调用的主流程实际上已经执行结束，只是要等待回调函数被调用。测试框架无法知道什么时候测试结束，因此需要有一个显式通知框架测试结束的环节，否则框架会一直等待（默认为 2000 毫秒）到超时报错。在回调函数中通过调用一个特殊的 done()函数来告知测试已经结束，它是作为 it 函数块测试函数的参数传入的。看下面的例子：

```
describe('Web 请求测试', function () {
   it('get 请求测试', function (done) {
      chai.request('http://www.example.com')
         .get('/resouces')
         .end(function(err, res){
            res.should.have.status(200);
            res.should.be.json;
            res.body.should.be.a('array');
            done();
      });
   });
});
```

有些异步操作的等待事件可能会超过系统默认的 2000 毫秒，此时可以使用命令行参数-t 或--timeout 重新设置默认超时时间，如下：

```
$ mocha -t 6000 example.test.js
```

13.3.2 chai

断言是判断代码的实际运行结果与预期是否一致的方法，在测试中使用断言是必不可少的，所有的测试用例（it 函数块）都应该含有一句或多句的断言。由于 Mocha 本身不含断言功能，所以需要引入断言库，在众多的断言库中，chai 因其链

式结构、仿自然语言语法和简单灵活的特性得到了大家的认可和广泛使用。

使用 chai 首先需要引入该模块：

```
import { expect } from 'chai';
```

先看一个断言的基本写法：

```
expect(add(1, 1)).to.be.equal(2);
```

expect 断言的写法基本都是一样的结构。头部是 expect 方法，尾部是断言方法，比如 equal、a/an、ok、match 等。两者之间使用 to 或 to.be 连接。

如果 expect 断言不成立，就会抛出一个错误，导致该项测试不通过。反之，只要运行测试代码的过程中没有抛出错误，那么测试用例就通过。如：

```
it('总是通过的测试用例', function() {});
```

上面的这个测试用例内部没有代码，但由于没有抛出任何错误，仍然会通过。

以下是 expect 常用的主要断言方法：

- ok：检查是否为真。
- true：检查对象是否为真。
- to.be、to：连接两个方法的链式方法。
- not：链接一个否定的断言，如 expect(false).not.to.be(true)。
- a/an：检查类型（也适用于数组类型）。
- include/contain：检查数组或字符串是否包含某个元素。
- below/above：检查是否大于或者小于某个限定值。

在 chai 中进行 http 请求测试时，需要引入 chai-http 模块。对应上节示例的前面部分代码如下：

```
import chai from 'chai';
```

```
import chaiHttp from 'chai-http';
var should = chai.should();
chai.use(chaiHttp);
```

13.3.3 jsdom

jsdom 是用 JavaScript 编写的基于 Node.js 的 WHATWG DOM 和 HTML 解析器实现。使用 jsdom 可以让我们脱离浏览器环境模拟对 DOM 的操作，并验证 DOM 操作之后的结果。

安装 jsdom 时使用的命令行：

```
npm install jsdom
```

在模拟真实 DOM 环境时，首先要保证 window、document 和 navigator 对象必须存在。以下代码完成了这个初始化过程：

```
import jsdom from 'jsdom';

if (typeof document === 'undefined') {
  global.document            =            jsdom.jsdom('<!doctype
html><html><body></body></html>');
  global.window = document.defaultView;
  global.navigator = global.window.navigator;
}
```

这个初始化过程与测试逻辑无关，通常作为一个独立的模块（如 setup.js）进行加载。结合 Mocha 使用时可利用 Mocha 命令的--require 参数进行加载。如下：

```
mocha --require ./test/setup.js
```

如果结合 npm 使用，需要修改 package.json 文件内容，在其中加入 test 命令片段：

```
{
  "scripts": {
```

```
      "test": "mocha --compilers js:babel-core/register --require./
test/setup.js"
    }
  }
```

每次运行 npm test 时，setup.js 就会包含在测试脚本中自动执行。

关于 jsdom 的具体资源可参看 https://github.com/tmpvar/jsdom，本书不作深入介绍。

13.3.4 react-addons-test-utils

react-addons-test-utils 工具对应 React.addons.TestUtils 命名空间下的一系列函数，该工具主要为 React 组件测试提供了一系列的 API。

1．模拟

```
Simulate.{eventName}(DOMElement element, object eventData)
```

模拟事件在 DOM 节点上派发，附带可选的 eventData 事件数据。这估计是 ReactTestUtils 中最有用的工具。

使用示例：

```
var node = this.refs.input.getDOMNode();
React.addons.TestUtils.Simulate.click(node);
React.addons.TestUtils.Simulate.change(node, {target: {value:
'Hello, world'}});
React.addons.TestUtils.Simulate.keyDown(node, {key: "Enter"});
```

Simulate 对 React 能识别的每个事件都提供了对应的成员函数，以激发响应的事件。

2．renderIntoDocument()

```
ReactComponent renderIntoDocument(ReactComponent instance)
```

将组件渲染到分离（还未挂接到 Document）的 DOM 节点。这个函数需要有 DOM 环境，即存在全局可用的 window、window.document 和 window.document.createElement。

3．mockComponent()

```
object mockComponent(function componentClass, [string mockTagName])
```

改变指定组件的渲染结构，用一个简单的 （或者是 mockTagName 指定的其他标签）来包含该组件的任何子节点。如果将组件比作一个包裹，该函数就是将包裹更换为 div 或指定的标签。常用于测试子节点。

4．isElementOfType()

```
boolean isElementOfType(ReactElement element, function componentClass)
```

如果 element 是一个类型为 componentClass 的 React 元素，则返回 true。

5．isDOMComponent()

```
boolean isDOMComponent(ReactComponent instance)
```

如果 instance 是一个 DOM 组件（如 或），则返回 true。

6．isCompositeComponent()

```
boolean isCompositeComponent(ReactComponent instance)`
```

如果 instance 是一个自定义组件（通过 React.createClass()创建），则返回 true。

7．isCompositeComponentWithType()

```
boolean isCompositeComponentWithType(ReactComponent instance, function
componentClass)
```

如果 instance 是一个自定义的组件（通过 React.createClass()创建），且此组件的类型是 componentClass，则返回 true，否则返回 false。

8．findAllInRenderedTree()

```
array findAllInRenderedTree(ReactComponent tree, function test)
```

遍历 tree 组件下的所有组件，搜集调用 test(component)后返回值为 true 的所有组件。这个函数可以为其他测试提供原始数据。

9．scryRenderedDOMComponentsWithClass()

```
array scryRenderedDOMComponentsWithClass(ReactComponent tree, string className)
```

查找 tree 组件下带有 className 类名的所有 DOM 组件实例。

10．findRenderedDOMComponentWithClass()

```
ReactComponent findRenderedDOMComponentWithClass(ReactComponent tree, string className)
```

类似于 scryRenderedDOMComponentsWithClass()，但是它只返回一个组件实例，如果有多个组件实例，则会抛出异常。

11．scryRenderedDOMComponentsWithTag()

```
array scryRenderedDOMComponentsWithTag(ReactComponent tree, string tagName)
```

在渲染后的 tree 组件实例树中找出所有标签名字符合 tagName 的 DOM 组件实例。

12．findRenderedDOMComponentWithTag()

```
ReactComponent findRenderedDOMComponentWithTag(ReactComponent tree, string tagName)
```

类似于 scryRenderedDOMComponentsWithTag()，但是它只返回一个结果，如果有其他满足条件的结果，则会抛出异常。

13．scryRenderedComponentsWithType()

```
    array scryRenderedComponentsWithType(ReactComponent tree, function
componentClass)
```

找出所有组件实例，这些组件的类型为 componentClass。

14．findRenderedComponentWithType()

```
    ReactComponent findRenderedComponentWithType(ReactComponent tree,
function componentClass)
```

类似于 scryRenderedComponentsWithType()，但是它只返回一个结果，如果有其他满足条件的结果，则会抛出异常。

13.4 创建测试环境

典型的开发环境通常包含测试环节，一般，在项目根目录下有 test 文件夹专门用于存放测试代码，其中还包括 jsdom 的初始化文件 setup.js，并在 package.json 文件中的 scripts 节配置：

```
    "test": "mocha --compilers js:babel-core/register --require. /test/
setup.js"
```

通过 npm test 启动测试过程，运行完毕后会给出测试报告。

13.5 React 组件测试

在针对 React 组件进行测试时，情况有点特殊，因为一个 React 组件的渲染过程包含虚拟 DOM 对象和真实浏览器 DOM 节点两个环节。在实际测试时，可能这两个环节都需要进行测试。因此，官方测试工具库对这两种形式都提供了对应的测试解决方案。

- 浅渲染（Shallow Rendering）：测试虚拟 DOM 的方法。
- DOM 渲染（DOM Rendering）：测试真实 DOM 的方法。

本节所给出的绝大部分测试代码示例是针对前面 7.6 节的综合性实例所进行的测试，工程源代码在本书示例 chapter13/example-testing 文件夹下。

13.5.1 浅渲染

浅渲染将组件渲染成虚拟 DOM 对象，但与通常的渲染过程不同的是，它只渲染第一层，而不涉及所有子组件，因此处理速度很快。浅渲染不需要真实浏览器 DOM 环境，也无法实现与 DOM 互动。

浅渲染使用 react-addons-test-utils 模块的 createRenderer()函数创建一个渲染器，渲染器渲染组件并缓存渲染出的虚拟 DOM 节点，通过调用渲染器的 getRenderOutput()函数获得虚拟 DOM 对象。为方便起见，通常先实现一个通用的浅渲染函数 shallowRender()，该函数接受以一个组件类作为参数，返回经过浅渲染后对应的虚拟 DOM 对象，代码如下。

```
import TestUtils from 'react-addons-test-utils';
function shallowRender(Component) {
  const renderer = TestUtils.createRenderer();
```

```
    renderer.render(<Component/>);
    return renderer.getRenderOutput();
  }
```

下面编写第一个测试用例，用来测试 App 组件的标题是否正确。这个测试用例不涉及子组件，适合使用浅渲染测试。

```
  describe('浅渲染测试 1', function () {
    it('App 组件的标题应该是“消息列表”', function () {
      const app = shallowRender(App);
      expect(app.props.children[0].type).to.equal('h2');
      expect(app.props.children[0].props.children).to.equal('消息列表');
    });
  });
```

在上面的代码中，const app = shallowRender(App)表示对 App 组件进行“浅渲染”；app.props.children[0].props.children 是组件的标题。

第二个测试用例用来测试 MessageList 项的初始状态。

首先，需要修改 shallowRender()函数，让组件接收 props 数据。

```
  import TestUtils from 'react-addons-test-utils';
  function shallowRender(Component, props) {
    const renderer = TestUtils.createRenderer();
    renderer.render(<Component {...props}/>);
    return renderer.getRenderOutput();
  }
```

下面是测试用例代码。

```
  import MessageItem from '../app/components/MessageItem';
  describe('浅渲染测试 2', function () {
    it('消息条目初始状态为"未读"', function () {
      const messageItemData = { id: 0, name: '第一条消息', hasRead:
false };
```

```
      const messageItem = shallowRender(MessageItem, {message:message
ItemData});
      expect(messageItem.props.children[0].props.className.indexOf
('message-read'))
    .to.equal(-1);
     });
    });
```

在上面的代码中，MessageItem 是 App 的后代组件，对 App 浅渲染是不会渲染 MessageItem 的，因此浅渲染只能在 MessageItem 上调用。

13.5.2 全DOM渲染

官方测试工具库的第二种测试方法，是将组件渲染成真实的 DOM 节点，再进行测试。这种测试方法使用的是 renderIntoDocument()函数。

```
    import TestUtils from 'react-addons-test-utils';
    import App from '../app/components/App';
    const app = TestUtils.renderIntoDocument(<App/>);
```

renderIntoDocument 方法要求存在一个全局的真实 DOM 环境，否则无法运行。在实际测试时，使用 jsdom 模块模拟真实 DOM 环境，需要初始化 jsdom 的代码。

第三个测试用例是测试交互操作，即单击“删除”按钮的行为。

```
    describe('DOM 渲染测试 1', function () {
     it('单击“×”按钮，该条消息从列表中消失', function () {
       const app = TestUtils.renderIntoDocument(<App/>);
       let messageItems = TestUtils.scryRenderedDOMComponentsWithTag
(app, 'li');
       let messageCount = messageItems.length;
       let deleteButton = messageItems[0].querySelector('button');
       TestUtils.Simulate.click(deleteButton);
```

```
      let todoItemsAfterClick = TestUtils.scryRenderedDOMComponents
WithTag(app, 'li');
      expect(todoItemsAfterClick.length).to.equal(messageCount - 1);
    });
  });
```

在上面的代码中，首先将 App 渲染成真实的 DOM 节点，使用 scryRenderedDOMComponentsWithTag()函数找出 app 里面所有的 li 元素。然后，取出第一个 li 元素里面的 button 元素，使用 TestUtils.Simulate.click 方法在该元素上模拟用户单击。最后，判断剩下的 li 元素应该少了一个。

这种测试方法的基本流程，就是先找到目标节点再触发某种动作。除了 scryRenderedDOMComponentsWithTag()函数外，react-addons-test-utils 还提供了多种方法，帮助用户找到目标 DOM 节点。

13.5.3 使用findDOMNode方法查找DOM

要查找真实 DOM 节点，还可以使用 findDOMNode 方法，其最大优点是能够支持复杂的 CSS 选择器，更灵活。

使用这种方法来写第四个测试用例，即用户单击消息项本身时的行为。

```
  import {findDOMNode} from 'react-dom';
  describe('DOM 渲染测试 2', function (done) {
    it('单击消息条目，消息变为已读', function () {
      const app = TestUtils.renderIntoDocument(<App/>);
      const appDOM = findDOMNode(app);
      const messageItem = appDOM.querySelector('li:first-child span');
      let hasRead = messageItem.classList.contains('message-read');
      TestUtils.Simulate.click(messageItem);
      expect(messageItem.classList.contains('message-read')).to.be.
equal(!hasRead);
      TestUtils.Simulate.click(messageItem);
```

```
    });
  });
```

在上面的代码中，先调用 findDOMNode 方法获取 App 所在的 DOM 节点。然后，找出第一个 li 节点，并模拟在它上面的单击行为。最后，验证 classList 属性中的 message-read 是否出现或消失。

第五个测试用例是发送消息的功能测试。

```
  describe('DOM 渲染测试 3', function (done) {
    it('单击“发送”按钮，输入的消息出现在消息列表中', function () {
      const app = TestUtils.renderIntoDocument(<App/>);
      const appDOM = findDOMNode(app);
      let  messageItemsCount  =  appDOM.querySelectorAll('.message-
content').length;
      let addInput = appDOM.querySelector('input');
      addInput.value = '新增的消息';
      let addButton = appDOM.querySelector('.add-message button');
      TestUtils.Simulate.click(addButton);
      expect(appDOM.querySelectorAll('.message-content').length)
            .to.be.equal(messageItemsCount + 1);
    });
  });
```

在上面的代码中，先找到 input 输入框，添加一个值。然后，找到“发送”按钮，模拟用户对该按钮的单击行为。最后，验证新发送的消息项是否出现在消息列表中。

第四篇

React相关资源

React 发展得如火如荼，在开源社区已经出现了众多基于 React 进行开发的框架和产品。为便于读者掌控方向、减少时间成本、迎合未来趋势，本篇主要介绍其中的优秀开源产品，它们必将在未来产生更大的影响力。因篇幅所限，只对这些产品作一些简要介绍，其他相关资料读者可以自行查阅。

14

React相关资源介绍

本章主要介绍当前与 React 相关的一些著名开源组件资源。

14.1 React Starter Kit

React Starter Kit 是一个 Web 开发工具包，其官方网站为 https://www.reactstarterkit.com/。React Starter Kit 整合了 Node.js、Express、GraphQL 和 React 的开发，并集成了 webpack、Babel、Browsersync 三种主要的开发调试工具。使用 React Starter Kit 可以引导开发者通过简单的配置，创建出结构良好、易于使用的项目工程，非常适合初学者使用。

1．创建一个名为 MyApp 的项目

```
$ git clone -o react-starter-kit -b master --single-branch \
```

```
      https://github.com/kriasoft/react-starter-kit.git MyApp
$ cd MyApp
```

可见工程目录如下：

```
.
├─── /build/              # 编译输出目录
├─── /docs/               # 相关文档目录
├─── /node_modules/       # 第三方依赖和第三方工具
├─── /src/                # 源代码
│    ├─── /components/    # React 组件
│    ├─── /content/       # 静态页面
│    ├─── /core/          # Core framework and utility functions
│    ├─── /data/          # GraphQL 数据模型
│    ├─── /public/        # 静态文件，会被复制到/build/public
│    ├─── /routes/        # 路由信息
│    ├─── /client.js      # 客户端启动脚本
│    ├─── /config.js      # 全局设置
│    └─── /server.js      # 服务端启动脚本
├─── /test/               # 测试相关代码
├─── /tools/              # 自动构建工具
│    ├─── /lib/           # 工具库
│    ├─── /build.js       # 从源代码编译到 build 目录
│    ├─── /bundle.js      # 通过 webpack 打包资源
│    ├─── /clean.js       # 清空 build 文件夹
│    ├─── /copy.js        # 复制静态文件到 build 目录
│    ├─── /deploy.js      # 发布
│    ├─── /run.js         # 运行自动构建任务
│    ├─── /runServer.js   # 启动 Node.js 服务器
│    ├─── /start.js       # 启动开发服务器，带 live reload 功能
│    └─── /webpack.config.js  # webpack 配置文件
└─── package.json         # npm 配置文件
```

2. 安装所需依赖

在命令行中运行：

```
$ npm install
```

3. 运行

在命令行中运行：

```
$npm start
```

该命令会完成从 src 目录到 build 目录的构建，并启动 Node.js 服务器（node build/server.js）和 Browsersync。Browsersync 是一种与 live reload 类似，能让浏览器实时、快速响应文件更改（html、js、css、sass、less 等）并自动刷新页面的工具。其优点是不需要在浏览器安装插件，可以同时在 PC、平板、手机等设备进行调试。

此时可在浏览器访问如下网址：

- http://localhost:3000/ — Node.js server (build/server.js)。
- http://localhost:3000/graphql — GraphQL server and IDE。
- http://localhost:3001/ — BrowserSync proxy with HMR, React Hot Transform。
- http://localhost:3002/ — BrowserSync control panel (UI)。

之后，只要对/src 文件夹中的内容进行修改，webpack 将自动重新编译和打包资源，所有的变化浏览器将自动刷新。

默认情况下，npm start 在开发模式下运行，若在部署模式下运行可以在命令行输入：

```
$ npm start --release
```

与开发模式相比，在发布模式下，React Starter Kit 将对资源进行优化和压缩。

4．其他命令

仅进行构建（不启动开发服务器）：

```
$ npm run build
```

进行发布模式构建：

```
$ npm run build -- --release
```

构建后，通过 Node.js 运行 build 文件夹中的 server.js 文件，即可启动此项目。

运行语法检查：

```
$ npm run lint
```

运行单元测试：

```
$ npm test                # Run unit tests with Mocha
$ npm run test:watch      # Launch unit test runner and start watching for changes
```

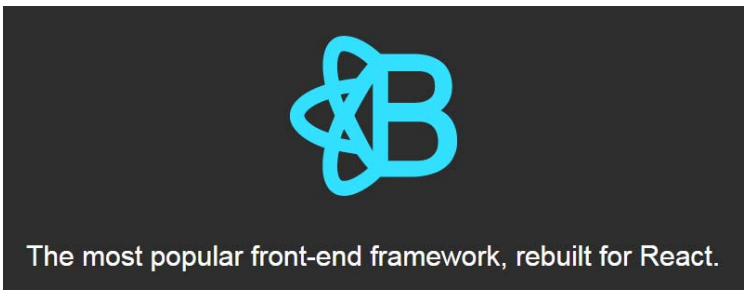
部署此项目（根据 tools/deploy.js 中的配置）：

```
$ npm run deploy
```

综上可知，React Starter Kit 是一个整合了诸多实用工具的快速开发环境，开发者不再需要根据不同工具的不同文档将这些工具配置在一起，免去了大部分烦琐的工具，开发者可以直接上手。同时，React Starter Kit 也支持根据不同需求进行定制，详情可参考官方网站。与 React Starter Kit 类似的还有 create-react-app（https://github.com/facebookincubator/create-react-app）及 react-boilerplate（http://reactboilerplate.com/）等。

14.2 React bootstrap

React bootstrap 是 Bootstrap 风格的 react 组件库。其官方网址是 http://react-bootstrap.github.io/。



1. 安装

使用 npm 进行安装，在命令行中运行：

```
$ npm install react-bootstrap --save
```

在使用中，React bootstrap 需要在 html 中引入 bootstrap.css 文件，若没有特定版本要求，建议使用最新版，开发者可以通过 cdn 将其加载到页面或者下载到本地作为静态文件发布。React bootstrap 只依赖 bootstrap 的 css 文件，并不需要依赖 bootstrap.js 和 jquery，这点需要注意。

2. 使用

如要使用相应的模块，需进行引入，例如使用 Button 时，需在代码中声明：

```
var Button = require('react-bootstrap').Button;
```

或：

```
var Alert = require('react-bootstrap').Alert;
```

这里推荐使用 ES 6 语法中的 import：

```
import Button from 'react-bootstrap/lib/Button';
```

或：

```
import { Button } from 'react-bootstrap';
```

下面以 Button 为例，显示三个不同样式的按钮，代码如下：

```
import {Button, ButtonToolbar} from 'react-bootstrap';

const buttonsInstance = (
  <ButtonToolbar>
    {/* 默认样式 */}
    <Button>Default</Button>
    {/* primary 样式，size 为 large */}
    <Button bsStyle="primary" bsSize="large">Primary</Button>
    {/* success 样式 */}
    <Button bsStyle="success">Success</Button>
  </ButtonToolbar>
);

ReactDOM.render(buttonsInstance,
document.getElementById('content'));
```

效果如下图所示，三个图标有样式和大小的区别。

对于按钮单击 onClick 事件，react-bootstrap 按钮只需声明 onClick 属性即可，示例如下：

```
var buttonsInstance = (
  <ButtonToolbar>
```

```
        <Button onClick={test}>Default</Button>
      </ButtonToolbar>
    );
    function showHi() {
        alert("hi")
    }
```

本书仅以 Button 为例作简要介绍，更多组件请访问 http://react-bootstrap.github.io/components.html，该项目仍在不断完善过程中，目前尚有很多 bootstrap 组件没有 React 实现。

14.3 Material-UI

Material-UI 是 Google Material Design 库的 React 实现，官方网站为 http://www.material- ui.com/。Google Material Design 是谷歌推出的全新设计理念，采用大胆的色彩、流畅的动画播放，以及卡片式的简洁设计。Material Design 风格的设计拥有干净的排版和简单的布局，有关 Material Design 的更多细节可参考官方网站 https: //material.google.com。

1. 安装

安装时直接按模块安装即可，命令如下：

```
$npm install material-ui
```

2. 使用

仍以 Button 为例，Material-UI 提供了多种按钮风格，下面以 RaisedButton 为例进行说明。

```
import RaisedButton from 'material-ui/RaisedButton';
import MuiThemeProvider from 'material-ui/styles/MuiThemeProvider';

const style = {
  margin: 12,
};

const App = () => (
  <MuiThemeProvider>
    <RaisedButtonExampleSimple />
  </MuiThemeProvider>
);

const RaisedButtonExampleSimple = () => (
  <div>
    <RaisedButton label="Default" style={style} />
    <RaisedButton label="Primary" primary={true} style={style} />
    <RaisedButton label="Secondary" secondary={true} style={style} />
    <RaisedButton label="Disabled" disabled={true} style={style} />
  </div>
);
ReactDOM.render(<App />, document.getElementById('content'));
```

效果如下图所示：

代码中的 MuiThemeProvider 用于载入不同的主题，这里仅使用默认主题，不做具体介绍。

相比于 react bootstrap，Material-UI 组件更加丰富，Material Design 中的大部分组件都已经有了 React 实现，而且文档完善，主题更方便定制，用例翔实，推荐使用。

14.4 Ant Design

Ant Design 是蚂蚁金服推出的一个中台设计语言。官方网站是 http://ant.design/。与前面介绍的几种资源不同，Ant Design 不仅仅是组件库，更是一种设计语言，旨在统一中台项目的前端 UI 设计，屏蔽不必要的设计差异和实现成本，解放设计和前端的研发资源。

对于中台设计语言，官方是这么描述的：

在中台产品的研发过程中，会出现不同的设计规范和实现方式，但往往存在很多类似的页面和组件，给设计师和工程师带来很多困扰和重复建设，大大降低了产品的研发效率。蚂蚁金服体验技术部经过大量的项目实践和总结，研发出一种中台设计语言——Ant Design，旨在统一中台项目的前端 UI 设计，屏蔽不必要的设计差异和实现成本，解放设计和前端的研发资源。Ant Design 是一个致力于提升用户和设计者使用体验的中台设计语言。它模糊了产品经理、交互设计师、视觉设计师、前端工程师、开发工程师等角色边界，将进行 UE 设计和 UI 设计的

人员统称为“设计者”，利用统一的规范进行设计赋能，全面提高中台产品体验和研发效率。

这里介绍的 Ant Design of React 是 Ant Design 的 React 实现，开发和服务于企业级后台产品。

1．特性

（1）提炼和服务企业级中后台产品的交互语言和视觉风格。

（2）在 React Component 基础上精心封装的高质量 UI 组件。

（3）基于 npm + webpack + babel 的工作流，支持 ES 6 和 TypeScript。

安装时直接运行下面的命令即可：

```
$ npm install antd
```

示例：

```
import { DatePicker } from 'antd';
ReactDOM.render(<DatePicker />, mountNode);
```

引入样式：

```
import 'antd/dist/antd.css';  // or 'antd/dist/antd.less'
```

按需加载可通过写法 import DatePicker from 'antd/lib/date-picker'，或使用插件 babel-plugin-antd（此插件支持 js 和 css 同时按需加载）实现。

2．开发工具

Ant Design 提供了 React 前端应用开发的脚手架工具，可以安装到全局直接使用。安装命令如下：

```
$ npm install antd-init -g
```

在空目录运行 antd-init 可以初始化一个 antd 的前端应用。

```
$ mkdir foo && cd foo
$ antd-init
```

要运行应用，输入以下命令：

```
$ npm start
```

可以启动开发服务器。

14.5 React-d 3 与echarts-for-react

D3 是前端使用非常多的图表工具，React-d3 是一个 D3 的 React 封装，官方网址为 https://reactiva.github.io/react-d3-website/。

安装方法也没有什么不同，命令如下：

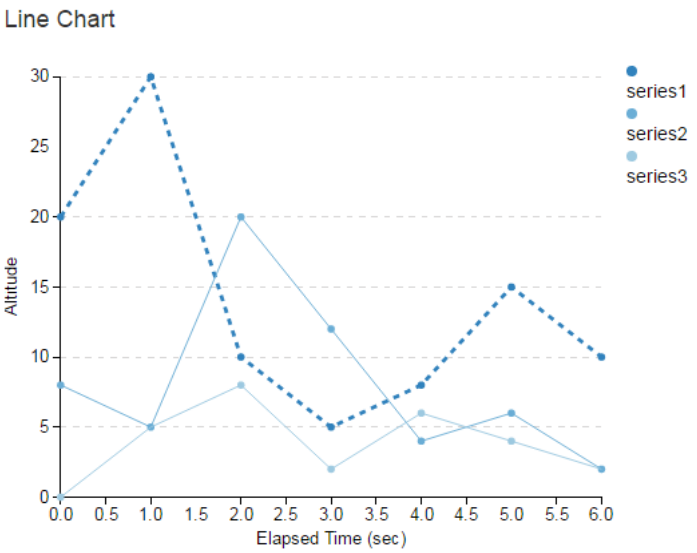
```
$ npm install react-d3
```

加载模块：

```
var rd3 = require('react-d3');
// es6
import rd3 from 'react-d3';
```

支持的图表类型有：

```
var BarChart = rd3.BarChart;
var LineChart = rd3.LineChart;
var PieChart = rd3.PieChart;
var AreaChart = rd3.AreaChart;
var Treemap = rd3.Treemap;
var ScatterChart = rd3.ScatterChart;
var CandleStickChart = rd3.CandleStickChart;
```



以 LineChart 为例：

```
var lineData = [{
    name: "series1",
    values: [ { x: 0, y: 20 }, ..., { x: 24, y: 10 } ],
    strokeWidth: 3,
    strokeDashArray: "5,5",
  },
  ....
  {
    name: "series2",
    values: [ { x: 70, y: 82 }, ..., { x: 76, y: 82 } ]
  }
];

<LineChart
  legend={true}
  data={lineData}
```

```
    width='100%'
    height={400}
    viewBoxObject={
      x: 0,
      y: 0,
      width: 500,
      height: 400
    }
    title="Line Chart"
    yAxisLabel="Altitude"
    xAxisLabel="Elapsed Time (sec)"
    gridHorizontal={true}
  />
```

Echarts-for-react 是一个对 Baidu Echarts 的 React 封装，习惯使用 Echarts 的用户可以通过它在 React 中使用 Echarts，网址为 http://git.hust.cc/echarts-for-react。

目前，在 React 中使用图表基本都是通过封装的方式来实现的，这也体现了 React 的开放性和模块化优势。通过封装，基本可以把任何已有的组件，封装成 React 组件，使其可在 React 中使用。

14.6 React Storybook

React Storybook 是一个 React 组件的 UI 开发环境，官方网站为 https://getstorybook.io。使用它可以实现组件状态的可视化交互式开发。

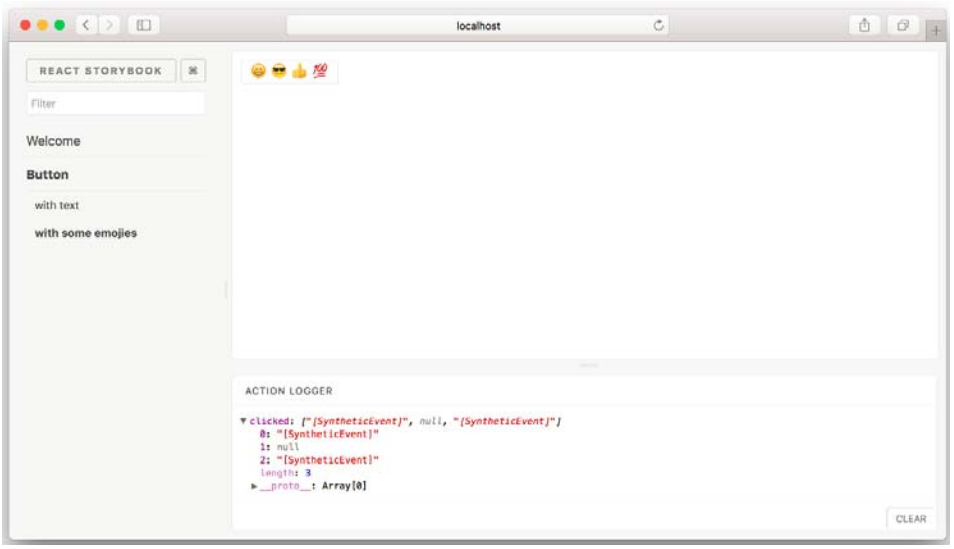
安装命令如下：

```
npm i -g getstorybook
cd my-react-app
getstorybook
```

要运行应用，输入以下命令：

```
npm run storybook
```

根据运行提示的地址，打开浏览器，即可看到它的页面。



使用时用户将自己的 React 组件放到指定的文件夹下，即可在页面中看到对应的组件，选中后可以看到组件的样子。在开发页面中可以对组件进行开发和测试，任何事件、动作响应都可以在这个界面中进行体现，非常方便调试。React Storybook 已经集成热替换功能，任何对组件的编辑，都可以及时更新到页面上。

14.7 awesome-react

awesome-react 是一个 React 资源列表，收录了上千资源信息，开发人员可以在上面找到相应的资源，网址为 https://github.com/enaqx/awesome-react。类似的资源列表还有 https://github.com/brillout/awesome-react-components。